Let's Build A Simple Interpreter. Part 1. (https://ruslanspivak.com/lsbasi-part1/)

Date Mon, June 15, 2015

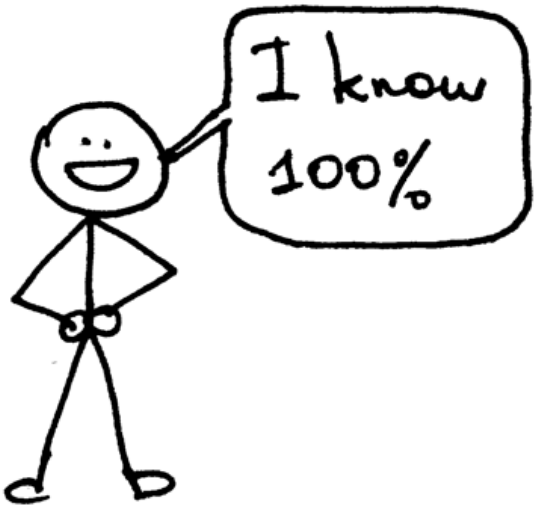
> ***"If you don't know how compilers work, then you don't know how computers work. If you're not 100% sure whether you know how compilers work, then you don't know how they work."*** — *Steve Yegge*

There you have it. Think about it. It doesn't really matter whether you're a newbie or a seasoned software developer: if you don't know how compilers and interpreters work, then you don't know how computers work. It's that simple.

So, do you know how compilers and interpreters work? And I mean, are you 100% sure that you know how they work? If you don't.

Or if you don't and you're really agitated about it.

Do not worry. If you stick around and work through the series and build an interpreter and a compiler with me you will know how they work in the end. And you will become a confident happy camper too. At least I hope so.



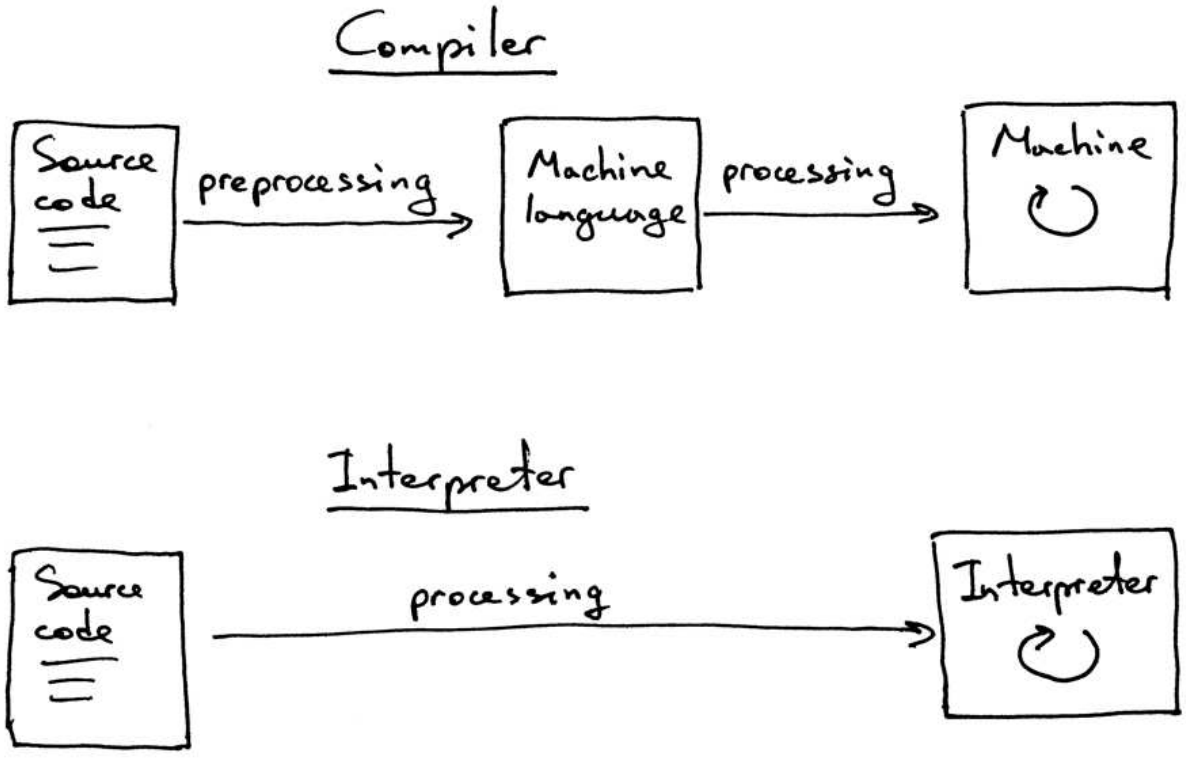
Why would you study interpreters and compilers? I will give you three reasons.

1. To write an interpreter or a compiler you have to have a lot of technical skills that you need to use together. Writing an interpreter or a compiler will help you improve those skills and become a better software developer. As well, the skills you will learn are useful in writing any software, not just interpreters or compilers.
2. You really want to know how computers work. Often interpreters and compilers look like magic. And you shouldn't be comfortable with that magic. You want to demystify the process of building an interpreter and a compiler, understand how they work, and get in control of things.
3. You want to create your own programming language or domain specific language. If you create one, you will also need to create either an interpreter or a compiler for it. Recently, there has been a resurgence of interest in new programming languages. And you can see a new programming language pop up almost every day: Elixir, Go, Rust just to name a few.

Okay, but what are interpreters and compilers?

The goal of an **interpreter** or a **compiler** is to translate a source program in some high-level language into some other form. Pretty vague, isn't it? Just bear with me, later in the series you will learn exactly what the source program is translated into.

At this point you may also wonder what the difference is between an interpreter and a compiler. For the purpose of this series, let's agree that if a translator translates a source program into machine language, it is a **compiler**. If a translator processes and executes the source program without translating it into machine language first, it is an **interpreter**. Visually it looks something like this:



I hope that by now you're convinced that you really want to study and build an interpreter and a compiler. What can you expect from this series on interpreters?

Here is the deal. You and I are going to create a simple interpreter for a large subset of Pascal (https://en.wikipedia.org/wiki/Pascal_%28programming_language%29) language. At the end of this series you will have a working Pascal interpreter and a source-level debugger like Python's pdb (https://docs.python.org/2/library/pdb.html).

You might ask, why Pascal? For one thing, it's not a made-up language that I came up with just for this series: it's a real programming language that has many important language constructs. And some old, but useful, CS books use Pascal programming language in their examples (I understand that that's not a particularly compelling reason to choose a language to build an interpreter for, but I thought it would be nice for a change to learn a non-mainstream language :)

Here is an example of a factorial function in Pascal that you will be able to interpret with your own interpreter and debug with the interactive source-level debugger that you will create along the way:

```
program factorial;

function factorial(n: integer): longint;
begin
    if n = 0 then
        factorial := 1
    else
        factorial := n * factorial(n - 1);
end;

var
    n: integer;

begin
    for n := 0 to 16 do
        writeln(n, '! = ', factorial(n));
end.
```

The implementation language of the Pascal interpreter will be Python, but you can use any language you want because the ideas presented don't depend on any particular implementation language. Okay, let's get down to business. Ready, set, go!

You will start your first foray into interpreters and compilers by writing a simple interpreter of arithmetic expressions, also known as a calculator. Today the goal is pretty minimalistic: to make your calculator handle the addition of two single digit integers like **3+5**. Here is the source code for your calculator, sorry, interpreter:

```
# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, EOF = 'INTEGER', 'PLUS', 'EOF'


class Token(object):
    def __init__(self, type, value):
        # token type: INTEGER, PLUS, or EOF
        self.type = type
        # token value: 0, 1, 2. 3, 4, 5, 6, 7, 8, 9, '+', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS '+')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Interpreter(object):
    def __init__(self, text):
        # client string input, e.g. "3+5"
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        # current token instance
        self.current_token = None

    def error(self):
        raise Exception('Error parsing input')

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        text = self.text

        # is self.pos index past the end of the self.text ?
        # if so, then return EOF token because there is no more
        # input left to convert into tokens
        if self.pos > len(text) - 1:
            return Token(EOF, None)

        # get a character at the position self.pos and decide
```

```
        # what token to create based on the single character
        current_char = text[self.pos]

        # if the character is a digit then convert it to
        # integer, create an INTEGER token, increment self.pos
        # index to point to the next character after the digit,
        # and return the INTEGER token
        if current_char.isdigit():
            token = Token(INTEGER, int(current_char))
            self.pos += 1
            return token

        if current_char == '+':
            token = Token(PLUS, current_char)
            self.pos += 1
            return token

        self.error()

    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.get_next_token()
        else:
            self.error()

    def expr(self):
        """expr -> INTEGER PLUS INTEGER"""
        # set current token to the first token taken from the input
        self.current_token = self.get_next_token()

        # we expect the current token to be a single-digit integer
        left = self.current_token
        self.eat(INTEGER)

        # we expect the current token to be a '+' token
        op = self.current_token
        self.eat(PLUS)

        # we expect the current token to be a single-digit integer
        right = self.current_token
        self.eat(INTEGER)
        # after the above call the self.current_token is set to
        # EOF token

        # at this point INTEGER PLUS INTEGER sequence of tokens
        # has been successfully found and the method can just
        # return the result of adding two integers, thus
        # effectively interpreting client input
        result = left.value + right.value
        return result


def main():
```

```
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break
        if not text:
            continue
        interpreter = Interpreter(text)
        result = interpreter.expr()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into *calc1.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part1/calc1.py). Before you start digging deeper into the code, run the calculator on the command line and see it in action. Play with it! Here is a sample session on my laptop (if you want to run the calculator under Python3 you will need to replace *raw_input* with *input*):

```
$ python calc1.py
calc> 3+4
7
calc> 3+5
8
calc> 3+9
12
calc>
```

For your simple calculator to work properly without throwing an exception, your input needs to follow certain rules:

- Only single digit integers are allowed in the input
- The only arithmetic operation supported at the moment is addition
- No whitespace characters are allowed anywhere in the input

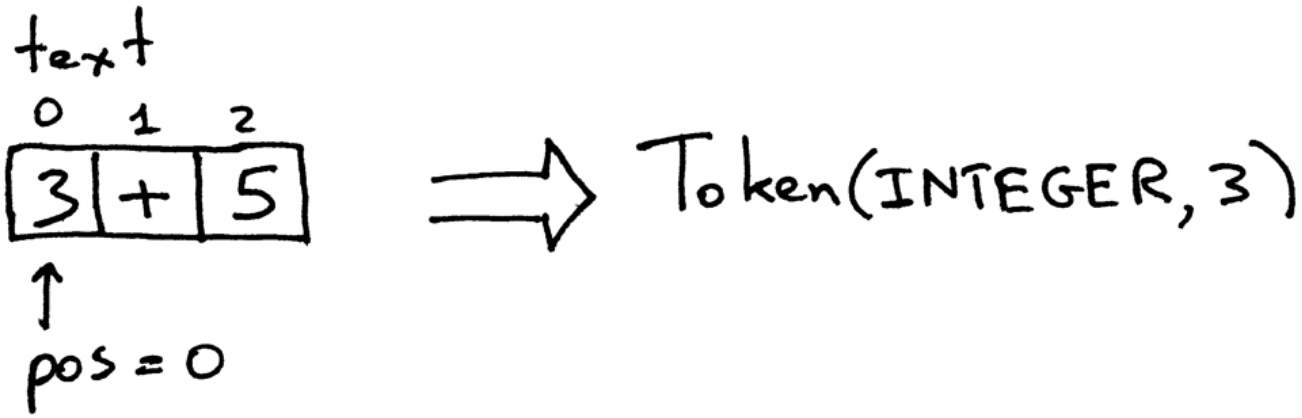
Those restrictions are necessary to make the calculator simple. Don't worry, you'll make it pretty complex pretty soon.

Okay, now let's dive in and see how your interpreter works and how it evaluates arithmetic expressions.

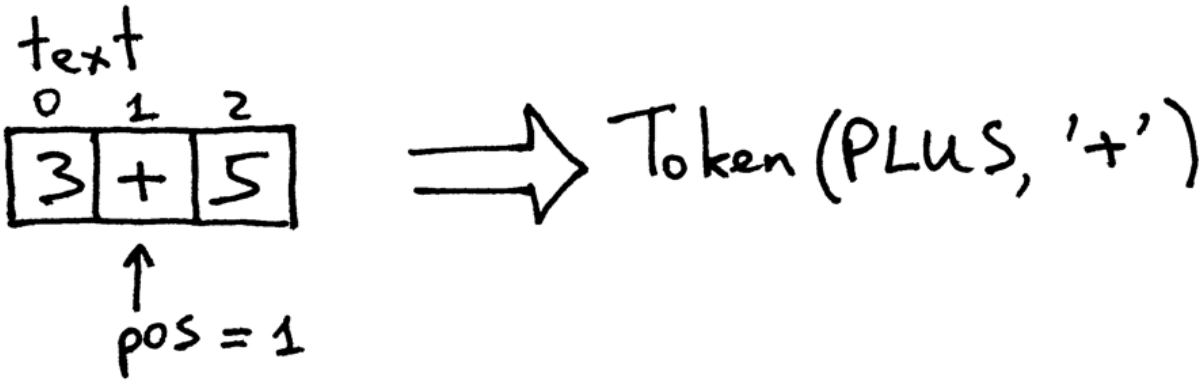
When you enter an expression *3+5* on the command line your interpreter gets a string *"3+5"*. In order for the interpreter to actually understand what to do with that string it first needs to break the input *"3+5"* into components called **tokens**. A **token** is an object that has a type and a value. For example, for the string *"3"* the type of the token will be INTEGER and the corresponding value will be integer *3*.

The process of breaking the input string into tokens is called **lexical analysis**. So, the first step your interpreter needs to do is read the input of characters and convert it into a stream of tokens. The part of the interpreter that does it is called a **lexical analyzer**, or **lexer** for short. You might also encounter other names for the same component, like **scanner** or **tokenizer**. They all mean the same: the part of your interpreter or compiler that turns the input of characters into a stream of tokens.

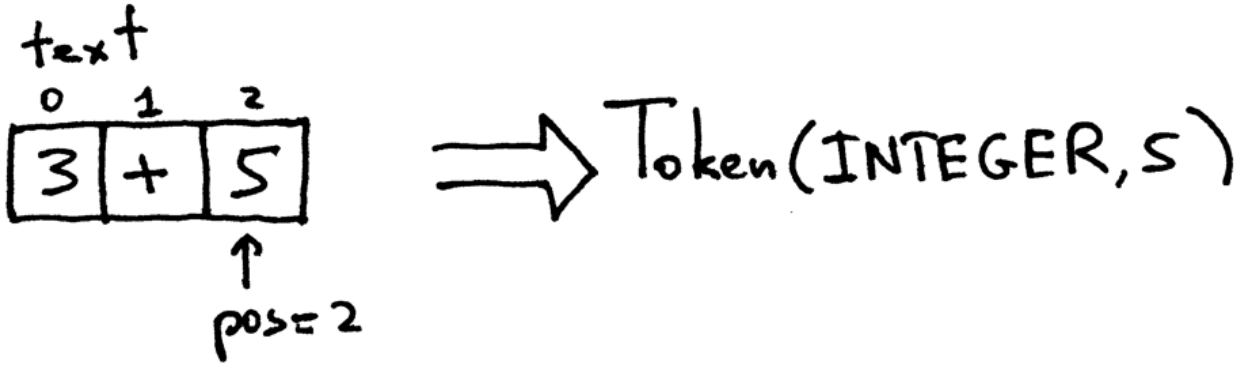
The method *get_next_token* of the *Interpreter* class is your lexical analyzer. Every time you call it, you get the next token created from the input of characters passed to the interpreter. Let's take a closer look at the method itself and see how it actually does its job of converting characters into tokens. The input is stored in the variable *text* that holds the input string and *pos* is an index into that string (think of the string as an array of characters). *pos* is initially set to 0 and points to the character *'3'*. The method first checks whether the character is a digit and if so, it increments *pos* and returns a token instance with the type INTEGER and the value set to the integer value of the string *'3'*, which is an integer *3*:



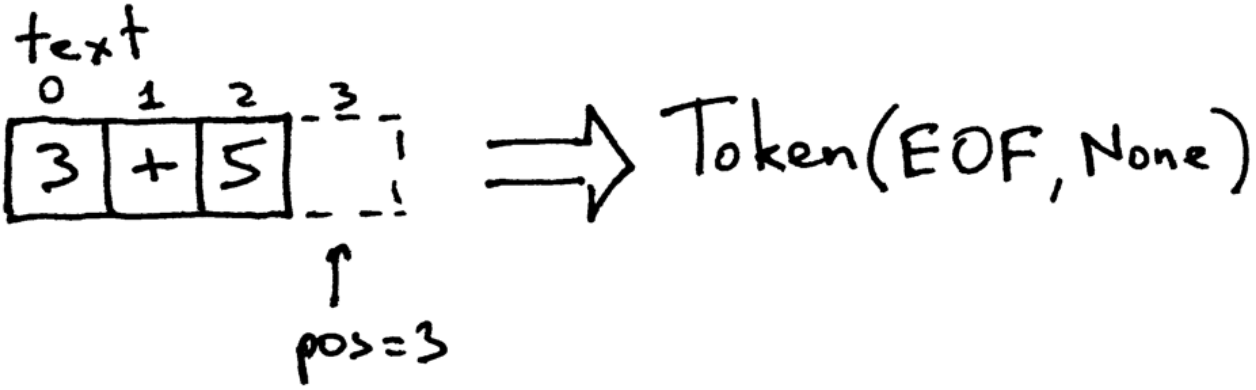
The *pos* now points to the *'+'* character in the *text*. The next time you call the method, it tests if a character at the position *pos* is a digit and then it tests if the character is a plus sign, which it is. As a result the method increments *pos* and returns a newly created token with the type PLUS and value *'+'*:



The *pos* now points to character *'5'*. When you call the *get_next_token* method again the method checks if it's a digit, which it is, so it increments *pos* and returns a new INTEGER token with the value of the token set to integer *5*:



Because the *pos* index is now past the end of the string *"3+5"* the *get_next_token* method returns the EOF token every time you call it:



Try it out and see for yourself how the lexer component of your calculator works:

```
>>> from calc1 import Interpreter
>>>
>>> interpreter = Interpreter('3+5')
>>> interpreter.get_next_token()
Token(INTEGER, 3)
>>>
>>> interpreter.get_next_token()
Token(PLUS, '+')
>>>
>>> interpreter.get_next_token()
Token(INTEGER, 5)
>>>
>>> interpreter.get_next_token()
Token(EOF, None)
>>>
```

So now that your interpreter has access to the stream of tokens made from the input characters, the interpreter needs to do something with it: it needs to find the structure in the flat stream of tokens it gets from the lexer *get_next_token*. Your interpreter expects to find the following structure in that stream: INTEGER -> PLUS -> INTEGER. That is, it tries to find a sequence of tokens: integer followed by a plus sign followed by an integer.

The method responsible for finding and interpreting that structure is *expr*. This method verifies that the sequence of tokens does indeed correspond to the expected sequence of tokens, i.e INTEGER -> PLUS -> INTEGER. After it's successfully confirmed the structure, it generates the result by adding the value of the token on the left side of the PLUS and the right side of the PLUS, thus successfully interpreting the arithmetic expression you passed to the interpreter.

The *expr* method itself uses the helper method *eat* to verify that the token type passed to the *eat* method matches the current token type. After matching the passed token type the *eat* method gets the next token and assigns it to the *current_token* variable, thus effectively "eating" the currently matched token and advancing the imaginary pointer in the stream of tokens. If the structure in the stream of tokens doesn't correspond to the expected INTEGER PLUS INTEGER sequence of tokens the *eat* method throws an exception.

Let's recap what your interpreter does to evaluate an arithmetic expression:

- The interpreter accepts an input string, let's say "3+5"

- The interpreter calls the *expr* method to find a structure in the stream of tokens returned by the lexical analyzer *get_next_token*. The structure it tries to find is of the form INTEGER PLUS INTEGER. After it's confirmed the structure, it interprets the input by adding the values of two INTEGER tokens because it's clear to the interpreter at that point that what it needs to do is add two integers, 3 and 5.

Congratulate yourself. You've just learned how to build your very first interpreter!

Now it's time for exercises.

You didn't think you would just read this article and that would be enough, did you? Okay, get your hands dirty and do the following exercises:

1. Modify the code to allow multiple-digit integers in the input, for example "12+3"
2. Add a method that skips whitespace characters so that your calculator can handle inputs with whitespace characters like " 12 + 3"
3. Modify the code and instead of '+' handle '-' to evaluate subtractions like "7-5"

Check your understanding

1. What is an interpreter?
2. What is a compiler?
3. What's the difference between an interpreter and a compiler?
4. What is a token?
5. What is the name of the process that breaks input apart into tokens?
6. What is the part of the interpreter that does lexical analysis called?
7. What are the other common names for that part of an interpreter or a compiler?

Before I finish this article, I really want you to commit to studying interpreters and compilers. And I want you to do it right now. Don't put it on the back burner. Don't wait. If you've skimmed the article, start over. If you've read it carefully but haven't done exercises - do them now. If you've done only some of them, finish the rest. You get the idea. And you know what? Sign the commitment pledge to start learning about interpreters and compilers today!

I, _____, of being sound mind and body, do hereby pledge to commit to studying interpreters and compilers starting today and get to a point where I know 100% how they work!

Signature:

Date:

Sign it, date it, and put it somewhere where you can see it every day to make sure that you stick to your commitment. And keep in mind the definition of commitment:

> "Commitment is doing the thing you said you were going to do long after the mood you said it in has left you." — Darren Hardy

Okay, that's it for today. In the next article of the mini series you will extend your calculator to handle more arithmetic expressions. Stay tuned.

If you can't wait for the second article and are chomping at the bit to start digging deeper into interpreters and compilers, here is a list of books I recommend that will help you along the way:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

Let's Build A Simple Interpreter. Part 2. (https://ruslanspivak.com/lsbasi-part2/)

Date Fri, July 03, 2015

In their amazing book "The 5 Elements of Effective Thinking" the authors Burger and Starbird share a story about how they observed Tony Plog, an internationally acclaimed trumpet virtuoso, conduct a master class for accomplished trumpet players. The students first played complex music phrases, which they played perfectly well. But then they were asked to play very basic, simple notes. When they played the notes, the notes sounded childish compared to the previously played complex phrases. After they finished playing, the master teacher also played the same notes, but when he played them, they did not sound childish. The difference was stunning. Tony explained that mastering the performance of simple notes allows one to play complex pieces with greater control. The lesson was clear - to build true virtuosity one must focus on mastering simple, basic ideas.[1]

The lesson in the story clearly applies not only to music but also to software development. The story is a good reminder to all of us to not lose sight of the importance of deep work on simple, basic ideas even if it sometimes feels like a step back. While it is important to be proficient with a tool or framework you use, it is also extremely important to know the principles behind them. As Ralph Waldo Emerson said:

> *"If you learn only methods, you'll be tied to your methods. But if you learn principles, you can devise your own methods."*

On that note, let's dive into interpreters and compilers again.

Today I will show you a new version of the calculator from Part 1 (http://ruslanspivak.com/lsbasi-part1/) that will be able to:

1. Handle whitespace characters anywhere in the input string
2. Consume multi-digit integers from the input
3. Subtract two integers (currently it can only add integers)

Here is the source code for your new version of the calculator that can do all of the above:

```
# Token types
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, MINUS, EOF = 'INTEGER', 'PLUS', 'MINUS', 'EOF'


class Token(object):
    def __init__(self, type, value):
        # token type: INTEGER, PLUS, MINUS, or EOF
        self.type = type
        # token value: non-negative integer value, '+', '-', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS '+')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Interpreter(object):
    def __init__(self, text):
        # client string input, e.g. "3 + 5", "12 - 5", etc
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        # current token instance
        self.current_token = None
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Error parsing input')

    def advance(self):
        """Advance the 'pos' pointer and set the 'current_char' variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
```

```
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '+':
                self.advance()
                return Token(PLUS, '+')

            if self.current_char == '-':
                self.advance()
                return Token(MINUS, '-')

            self.error()

        return Token(EOF, None)

    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.get_next_token()
        else:
            self.error()

    def expr(self):
        """Parser / Interpreter

        expr -> INTEGER PLUS INTEGER
        expr -> INTEGER MINUS INTEGER
        """
        # set current token to the first token taken from the input
        self.current_token = self.get_next_token()

        # we expect the current token to be an integer
        left = self.current_token
        self.eat(INTEGER)

        # we expect the current token to be either a '+' or '-'
        op = self.current_token
```

```
        if op.type == PLUS:
            self.eat(PLUS)
        else:
            self.eat(MINUS)

        # we expect the current token to be an integer
        right = self.current_token
        self.eat(INTEGER)
        # after the above call the self.current_token is set to
        # EOF token

        # at this point either the INTEGER PLUS INTEGER or
        # the INTEGER MINUS INTEGER sequence of tokens
        # has been successfully found and the method can just
        # return the result of adding or subtracting two integers,
        # thus effectively interpreting client input
        if op.type == PLUS:
            result = left.value + right.value
        else:
            result = left.value - right.value
        return result


def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break
        if not text:
            continue
        interpreter = Interpreter(text)
        result = interpreter.expr()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into the *calc2.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part2/calc2.py). Try it out. See for yourself that it works as expected: it can handle whitespace characters anywhere in the input; it can accept multi-digit integers, and it can also subtract two integers as well as add two integers.

Here is a sample session that I ran on my laptop:

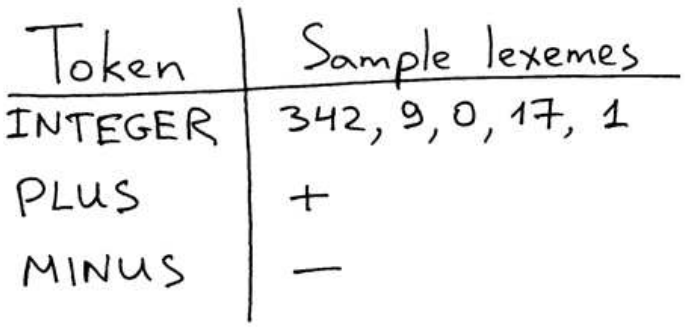
```
$ python calc2.py
calc> 27 + 3
30
calc> 27 - 7
20
calc>
```

The major code changes compared with the version from Part 1 (http://ruslanspivak.com/lsbasi-part1/) are:

1. The *get_next_token* method was refactored a bit. The logic to increment the *pos* pointer was factored into a separate method *advance*.
2. Two more methods were added: *skip_whitespace* to ignore whitespace characters and *integer* to handle multi-digit integers in the input.
3. The *expr* method was modified to recognize INTEGER -> MINUS -> INTEGER phrase in addition to INTEGER -> PLUS -> INTEGER phrase. The method now also interprets both addition and subtraction after having successfully recognized the corresponding phrase.

In Part 1 (http://ruslanspivak.com/lsbasi-part1/) you learned two important concepts, namely that of a **token** and a **lexical analyzer**. Today I would like to talk a little bit about **lexemes**, **parsing**, and **parsers**.

You already know about tokens. But in order for me to round out the discussion of tokens I need to mention lexemes. What is a lexeme? A **lexeme** is a sequence of characters that form a token. In the following picture you can see some examples of tokens and sample lexemes and hopefully it will make the relationship between them clear:

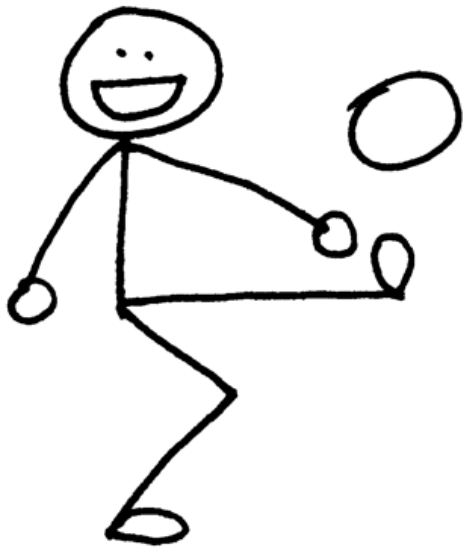


Now, remember our friend, the *expr* method? I said before that that's where the interpretation of an arithmetic expression actually happens. But before you can interpret an expression you first need to recognize what kind of phrase it is, whether it is addition or subtraction, for example. That's what the *expr* method essentially does: it finds the structure in the stream of tokens it gets from the *get_next_token* method and then it interprets the phrase that is has recognized, generating the result of the arithmetic expression.

The process of finding the structure in the stream of tokens, or put differently, the process of recognizing a phrase in the stream of tokens is called **parsing**. The part of an interpreter or compiler that performs that job is called a **parser**.

So now you know that the *expr* method is the part of your interpreter where both **parsing** and **interpreting** happens - the *expr* method first tries to recognize (**parse**) the INTEGER -> PLUS -> INTEGER or the INTEGER -> MINUS -> INTEGER phrase in the stream of tokens and after it has successfully recognized (**parsed**) one of those phrases, the method interprets it and returns the result of either addition or subtraction of two integers to the caller.

And now it's time for exercises again.



1. Extend the calculator to handle multiplication of two integers
2. Extend the calculator to handle division of two integers
3. Modify the code to interpret expressions containing an arbitrary number of additions and subtractions, for example "9 - 5 + 3 + 11"

Check your understanding.

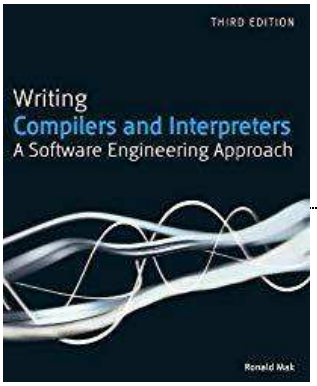
1. What is a lexeme?
2. What is the name of the process that finds the structure in the stream of tokens, or put differently, what is the name of the process that recognizes a certain phrase in that stream of tokens?
3. What is the name of the part of the interpreter (compiler) that does parsing?

I hope you liked today's material. In the next article of the series you will extend your calculator to handle more complex arithmetic expressions. Stay tuned.

And here is a list of books I recommend that will help you in your study of interpreters and compilers:

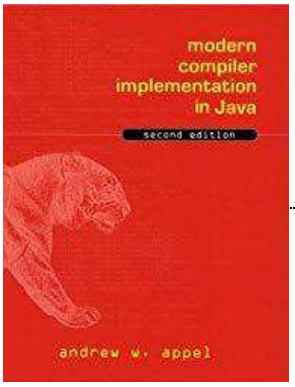
1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)



(http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=FYAZBCVOB66PGR6J)

3. Modern Compiler Implementation in Java (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=ZSKKZMV7YWR22NMW)

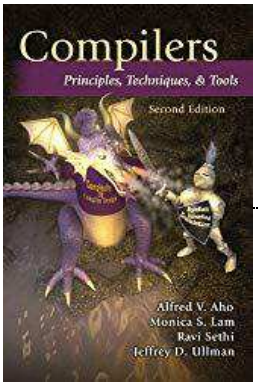


(http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=GPMSWTZYFC2M6MJE)

4. Modern Compiler Design (http://www.amazon.com/gp/product/1461446988/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1461446988&linkCode=as2&tag=russblo0b-20&linkId=PAXWJP5WCPZ7RKRD)

(http://www.amazon.com/gp/product/1461446988/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1461446988&linkCode=as2&tag=russblo0b-20&linkId=DZVYHZHDHYAPOQOD)

5. Compilers: Principles, Techniques, and Tools (2nd Edition) (http://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=GOEGDQG4HIHU56FQ)



(http://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=MD7L2CQHFXDYKOG6)

BTW, I'm writing a book **"Let's Build A Web Server: First Steps"** that explains how to write a basic web server from scratch. You can get a feel for the book here (http://ruslanspivak.com/lsbaws-part1/), here (http://ruslanspivak.com/lsbaws-part2/), and here (http://ruslanspivak.com/lsbaws-part3/). Subscribe to the mailing list to get the latest updates about the book and the release date.

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series:

- Let's Build A Simple Interpreter. Part 1. (/lsbasi-part1/)
- Let's Build A Simple Interpreter. Part 2. (/lsbasi-part2/)
- Let's Build A Simple Interpreter. Part 3. (/lsbasi-part3/)
- Let's Build A Simple Interpreter. Part 4. (/lsbasi-part4/)
- Let's Build A Simple Interpreter. Part 5. (/lsbasi-part5/)
- Let's Build A Simple Interpreter. Part 6. (/lsbasi-part6/)
- Let's Build A Simple Interpreter. Part 7. (/lsbasi-part7/)
- Let's Build A Simple Interpreter. Part 8. (/lsbasi-part8/)
- Let's Build A Simple Interpreter. Part 9. (/lsbasi-part9/)
- Let's Build A Simple Interpreter. Part 10. (/lsbasi-part10/)
- Let's Build A Simple Interpreter. Part 11. (/lsbasi-part11/)
- Let's Build A Simple Interpreter. Part 12. (/lsbasi-part12/)
- Let's Build A Simple Interpreter. Part 13. (/lsbasi-part13/)
- Let's Build A Simple Interpreter. Part 14. (/lsbasi-part14/)

---

1. The 5 Elements of Effective Thinking (http://www.amazon.com/gp/product/0691156662/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0691156662&linkCode=as2&tag=russblo0b-20&linkId=B7GSVLONUPCIBIVY)

Comments

Let's Build A Simple Interpreter. Part 3. (https://ruslanspivak.com/lsbasi-part3/)

Date Wed, August 12, 2015

I woke up this morning and I thought to myself: "Why do we find it so difficult to learn a new skill?"

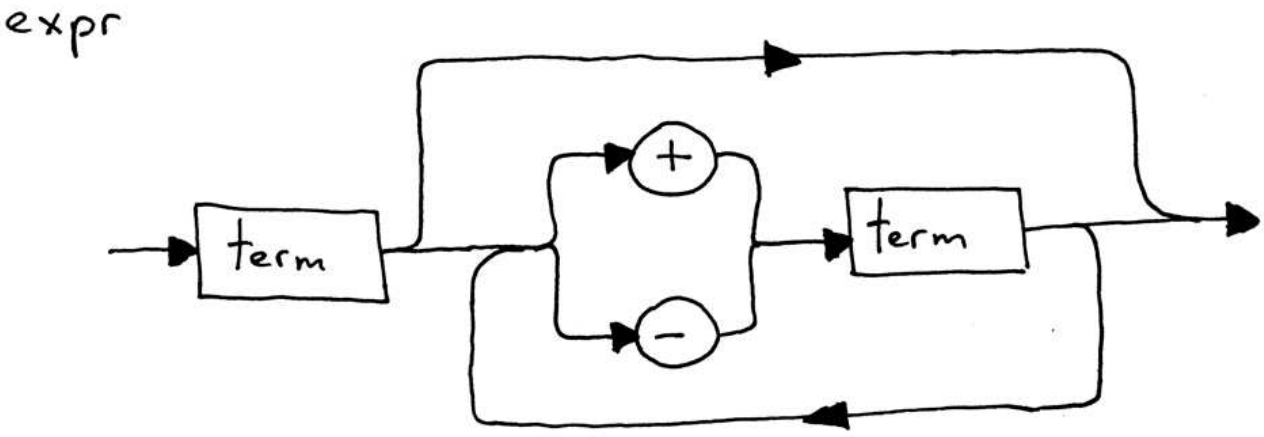
I don't think it's just because of the hard work. I think that one of the reasons might be that we spend a lot of time and hard work acquiring knowledge by reading and watching and not enough time translating that knowledge into a skill by practicing it. Take swimming, for example. You can spend a lot of time reading hundreds of books about swimming, talk for hours with experienced swimmers and coaches, watch all the training videos available, and you still will sink like a rock the first time you jump in the pool.

The bottom line is: it doesn't matter how well you think you know the subject - you have to put that knowledge into practice to turn it into a skill. To help you with the practice part I put exercises into Part 1 (http://ruslanspivak.com/lsbasi-part1/) and Part 2 (http://ruslanspivak.com/lsbasi-part2/) of the series. And yes, you will see more exercises in today's article and in future articles, I promise :)

Okay, let's get started with today's material, shall we?

So far, you've learned how to interpret arithmetic expressions that add or subtract two integers like "7 + 3" or "12 - 9". Today I'm going to talk about how to parse (recognize) and interpret arithmetic expressions that have any number of plus or minus operators in it, for example "7 - 3 + 2 - 1".

Graphically, the arithmetic expressions in this article can be represented with the following syntax diagram:



What is a syntax diagram? A **syntax diagram** is a graphical representation of a programming language's syntax rules. Basically, a syntax diagram visually shows you which statements are allowed in your programming language and which are not.

Syntax diagrams are pretty easy to read: just follow the paths indicated by the arrows. Some paths indicate choices. And some paths indicate loops.

You can read the above syntax diagram as following: a term optionally followed by a plus or minus sign, followed by another term, which in turn is optionally followed by a plus or minus sign followed by another term and so on. You get the picture, literally. You might wonder what a *"term"* is. For the purpose of this article a *"term"* is just an integer.

Syntax diagrams serve two main purposes:

- They graphically represent the specification (grammar) of a programming language.
- They can be used to help you write your parser - you can map a diagram to code by following simple rules.

You've learned that the process of recognizing a phrase in the stream of tokens is called **parsing**. And the part of an interpreter or compiler that performs that job is called a **parser**. Parsing is also called **syntax analysis**, and the parser is also aptly called, you guessed it right, a **syntax analyzer**.

According to the syntax diagram above, all of the following arithmetic expressions are valid:

- 3
- 3 + 4
- 7 - 3 + 2 - 1

Because syntax rules for arithmetic expressions in different programming languages are very similar we can use a Python shell to "test" our syntax diagram. Launch your Python shell and see for yourself:

```
>>> 3
3
>>> 3 + 4
7
>>> 7 - 3 + 2 - 1
5
```

No surprises here.

The expression "3 + " is not a valid arithmetic expression though because according to the syntax diagram the plus sign must be followed by a *term* (integer), otherwise it's a syntax error. Again, try it with a Python shell and see for yourself:

```
>>> 3 +
  File "<stdin>", line 1
    3 +
      ^
SyntaxError: invalid syntax
```

It's great to be able to use a Python shell to do some testing but let's map the above syntax diagram to code and use our own interpreter for testing, all right?

You know from the previous articles (Part 1 (http://ruslanspivak.com/lsbasi-part1/) and Part 2 (http://ruslanspivak.com/lsbasi-part2/)) that the *expr* method is where both our parser and interpreter live. Again, the parser just recognizes the structure making sure that it corresponds to some specifications and the interpreter actually evaluates the expression once the parser has successfully recognized (parsed) it.

The following code snippet shows the parser code corresponding to the diagram. The rectangular box from the syntax diagram (*term*) becomes a *term* method that parses an integer and the *expr* method just follows the syntax diagram flow:

```
def term(self):
    self.eat(INTEGER)

def expr(self):
    # set current token to the first token taken from the input
    self.current_token = self.get_next_token()

    self.term()
    while self.current_token.type in (PLUS, MINUS):
        token = self.current_token
        if token.type == PLUS:
            self.eat(PLUS)
            self.term()
        elif token.type == MINUS:
            self.eat(MINUS)
            self.term()
```

You can see that *expr* first calls the *term* method. Then the *expr* method has a *while* loop which can execute zero or more times. And inside the loop the parser makes a choice based on the token (whether it's a plus or minus sign). Spend some time proving to yourself that the code above does indeed follow the syntax diagram flow for arithmetic expressions.

The parser itself does not interpret anything though: if it recognizes an expression it's silent and if it doesn't, it throws out a syntax error. Let's modify the *expr* method and add the interpreter code:

```
def term(self):
    """Return an INTEGER token value"""
    token = self.current_token
    self.eat(INTEGER)
    return token.value

def expr(self):
    """Parser / Interpreter """
    # set current token to the first token taken from the input
    self.current_token = self.get_next_token()

    result = self.term()
    while self.current_token.type in (PLUS, MINUS):
        token = self.current_token
        if token.type == PLUS:
            self.eat(PLUS)
            result = result + self.term()
        elif token.type == MINUS:
            self.eat(MINUS)
            result = result - self.term()

    return result
```

Because the interpreter needs to evaluate an expression the *term* method was modified to return an integer value and the *expr* method was modified to perform addition and subtraction at the appropriate places and return the result of interpretation. Even though the code is pretty straightforward I recommend spending some time studying it.

Le's get moving and see the complete code of the interpreter now, okay?

Here is the source code for your new version of the calculator that can handle valid arithmetic expressions containing integers and any number of addition and subtraction operators:

```
# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, MINUS, EOF = 'INTEGER', 'PLUS', 'MINUS', 'EOF'


class Token(object):
    def __init__(self, type, value):
        # token type: INTEGER, PLUS, MINUS, or EOF
        self.type = type
        # token value: non-negative integer value, '+', '-', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS, '+')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Interpreter(object):
    def __init__(self, text):
        # client string input, e.g. "3 + 5", "12 - 5 + 3", etc
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        # current token instance
        self.current_token = None
        self.current_char = self.text[self.pos]

    ##########################################################
    # Lexer code                                             #
    ##########################################################
    def error(self):
        raise Exception('Invalid syntax')

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()
```

```
    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '+':
                self.advance()
                return Token(PLUS, '+')

            if self.current_char == '-':
                self.advance()
                return Token(MINUS, '-')

            self.error()

        return Token(EOF, None)

    ##########################################################
    # Parser / Interpreter code                              #
    ##########################################################
    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.get_next_token()
        else:
            self.error()

    def term(self):
        """Return an INTEGER token value."""
        token = self.current_token
        self.eat(INTEGER)
        return token.value

    def expr(self):
        """Arithmetic expression parser / interpreter."""
```

```
        # set current token to the first token taken from the input
        self.current_token = self.get_next_token()

        result = self.term()
        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
                result = result + self.term()
            elif token.type == MINUS:
                self.eat(MINUS)
                result = result - self.term()

        return result


def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break
        if not text:
            continue
        interpreter = Interpreter(text)
        result = interpreter.expr()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into the *calc3.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part3/calc3.py). Try it out. See for yourself that it can handle arithmetic expressions that you can derive from the syntax diagram I showed you earlier.

Here is a sample session that I ran on my laptop:

```
$ python calc3.py
calc> 3
3
calc> 7 - 4
3
calc> 10 + 5
15
calc> 7 - 3 + 2 - 1
5
calc> 10 + 1 + 2 - 3 + 4 + 6 - 15
5
calc> 3 +
Traceback (most recent call last):
  File "calc3.py", line 147, in <module>
    main()
  File "calc3.py", line 142, in main
    result = interpreter.expr()
  File "calc3.py", line 123, in expr
    result = result + self.term()
  File "calc3.py", line 110, in term
    self.eat(INTEGER)
  File "calc3.py", line 105, in eat
    self.error()
  File "calc3.py", line 45, in error
    raise Exception('Invalid syntax')
Exception: Invalid syntax
```

Remember those exercises I mentioned at the beginning of the article: here they are, as promised :)

- Draw a syntax diagram for arithmetic expressions that contain only multiplication and division, for example “7 * 4 / 2 * 3”. Seriously, just grab a pen or a pencil and try to draw one.
- Modify the source code of the calculator to interpret arithmetic expressions that contain only multiplication and division, for example “7 * 4 / 2 * 3”.
- Write an interpreter that handles arithmetic expressions like “7 - 3 + 2 - 1” from scratch. Use any programming language you’re comfortable with and write it off the top of your head without looking at the examples. When you do that, think about components involved: a *lexer* that takes an input and converts it into a stream of tokens, a *parser* that feeds off the stream of the tokens provided by the *lexer* and tries to recognize a structure in that stream, and an *interpreter* that generates results after the *parser* has successfully parsed (recognized) a valid arithmetic expression. String those

pieces together. Spend some time translating the knowledge you've acquired into a working interpreter for arithmetic expressions.

Check your understanding.

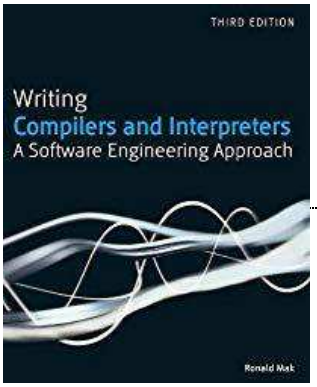
1. What is a syntax diagram?
2. What is syntax analysis?
3. What is a syntax analyzer?

Hey, look! You read all the way to the end. Thanks for hanging out here today and don't forget to do the exercises. :) I'll be back next time with a new article - stay tuned.

Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

   (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=FYAZBCVOB66PGR6J)
3. Modern Compiler Implementation in Java (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=ZSKKZMV7YWR22NMW)

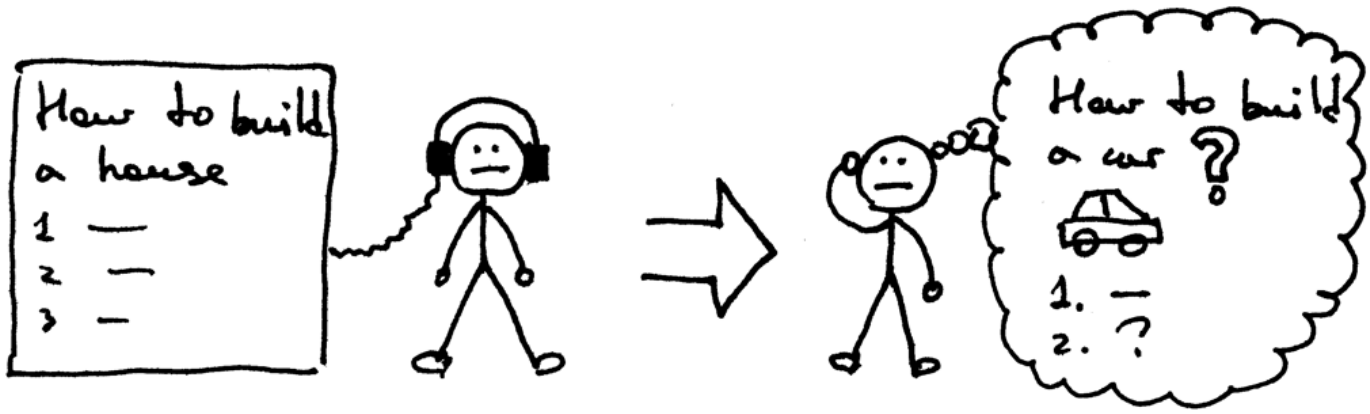
Let's Build A Simple Interpreter. Part 4. (https://ruslanspivak.com/lsbasi-part4/)

Date Fri, September 11, 2015

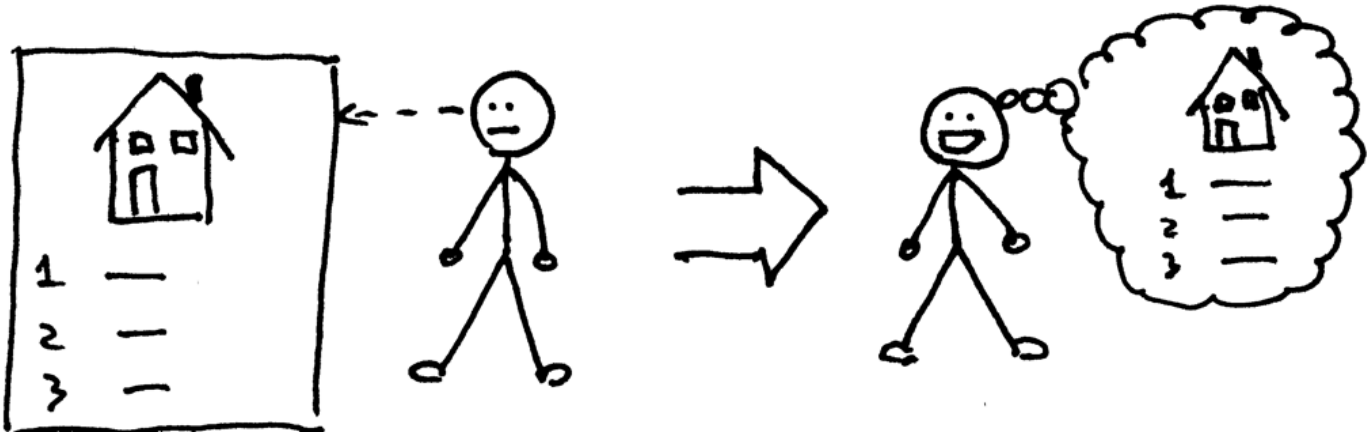
Have you been passively learning the material in these articles or have you been actively practicing it? I hope you've been actively practicing it. I really do :)

Remember what Confucius said?

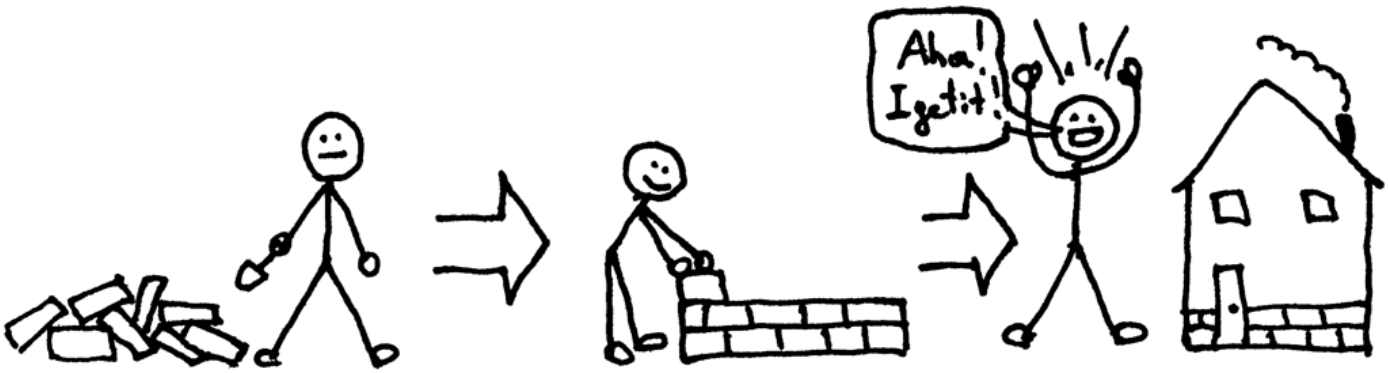
> *"I hear and I forget."*



> *"I see and I remember."*



> *"I do and I understand."*



In the previous article you learned how to parse (recognize) and interpret arithmetic expressions with any number of plus or minus operators in them, for example "7 - 3 + 2 - 1". You also learned about syntax diagrams and how they can be used to specify the syntax of a programming language.

Today you're going to learn how to parse and interpret arithmetic expressions with any number of multiplication and division operators in them, for example "7 * 4 / 2 * 3". The division in this article will be an integer division, so if the expression is "9 / 4", then the answer will be an integer: 2.

I will also talk quite a bit today about another widely used notation for specifying the syntax of a programming language. It's called ***context-free grammars*** (***grammars***, for short) or ***BNF*** (Backus-Naur Form). For the purpose of this article I will not use pure BNF (https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form) notation but more like a modified EBNF (https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form) notation.

Here are a couple of reasons to use grammars:

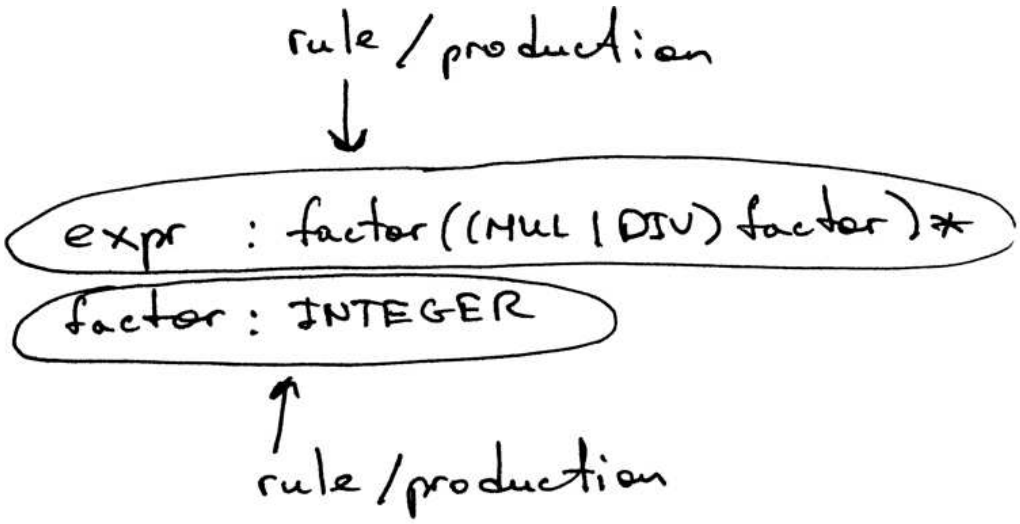
1. A grammar specifies the syntax of a programming language in a concise manner. Unlike syntax diagrams, grammars are very compact. You will see me using grammars more and more in future articles.
2. A grammar can serve as great documentation.
3. A grammar is a good starting point even if you manually write your parser from scratch. Quite often you can just convert the grammar to code by following a set of simple rules.
4. There is a set of tools, called *parser generators*, which accept a grammar as an input and automatically generate a parser for you based on that grammar. I will talk about those tools later on in the series.

Now, let's talk about the mechanical aspects of grammars, shall we?

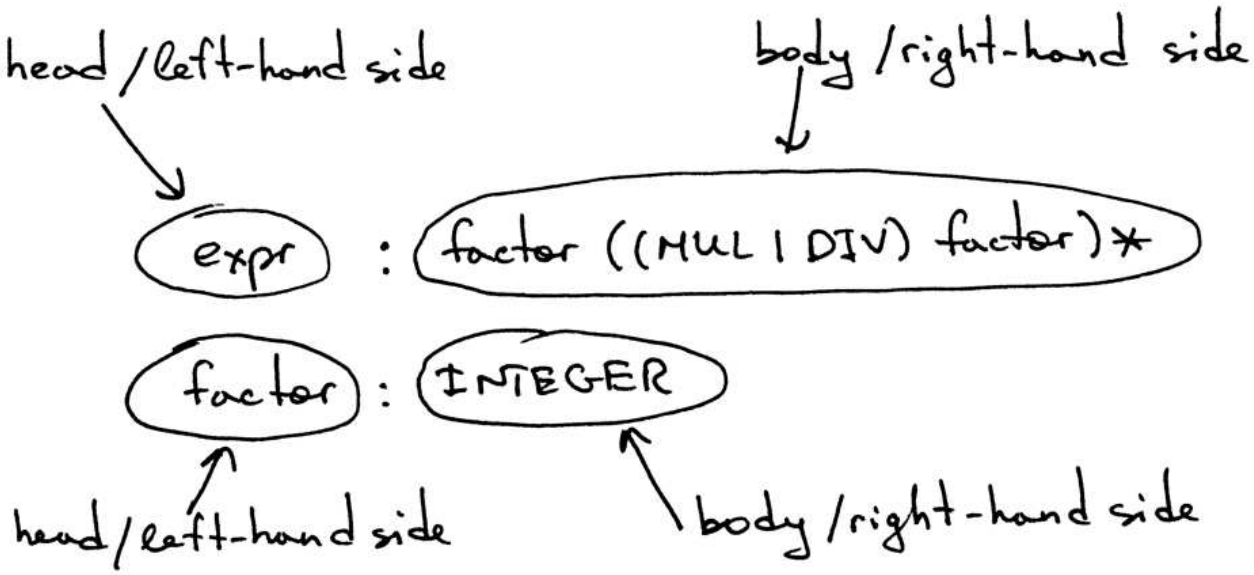
Here is a grammar that describes arithmetic expressions like "7 * 4 / 2 * 3" (it's just one of the many expressions that can be generated by the grammar):

```
expr   : factor ((MUL | DIV) factor)*
factor : INTEGER
```

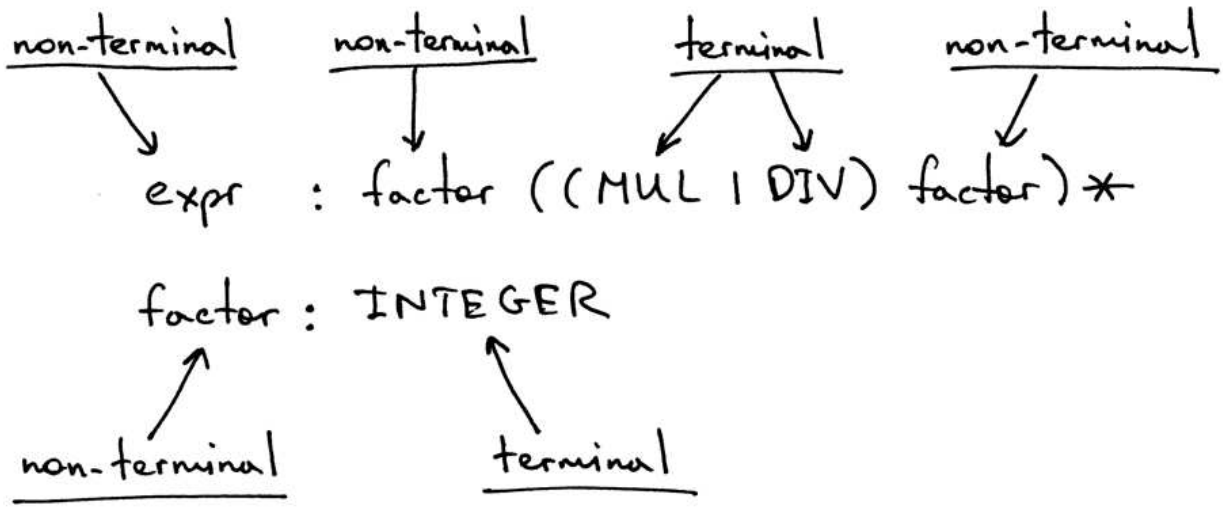
A grammar consists of a sequence of *rules*, also known as *productions*. There are two rules in our grammar:



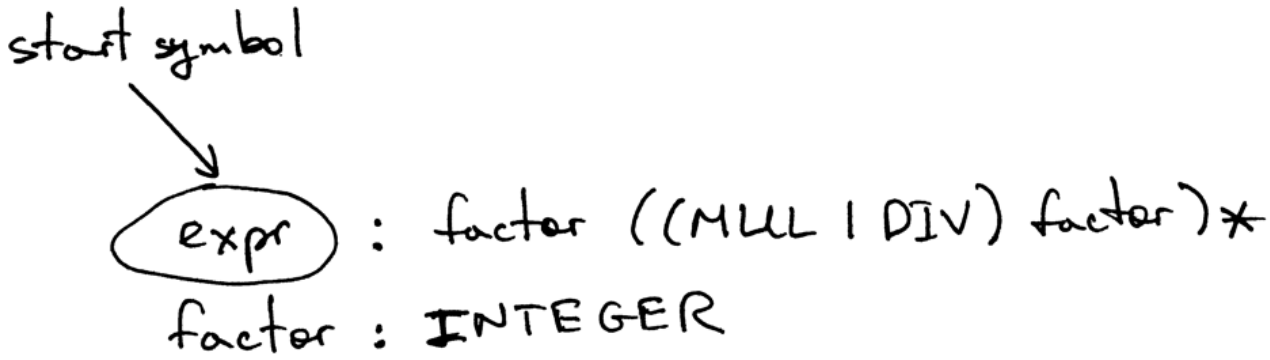
A rule consists of a *non-terminal*, called the ***head*** or ***left-hand side*** of the production, a colon, and a sequence of terminals and/or non-terminals, called the ***body*** or ***right-hand side*** of the production:



In the grammar I showed above, tokens like MUL, DIV, and INTEGER are called ***terminals*** and variables like *expr* and *factor* are called ***non-terminals***. Non-terminals usually consist of a sequence of terminals and/or non-terminals:



The non-terminal symbol on the left side of the first rule is called the ***start symbol***. In the case of our grammar, the start symbol is *expr*:



You can read the rule *expr* as "An *expr* can be a *factor* optionally followed by a *multiplication* or *division* operator followed by another *factor*, which in turn is optionally followed by a *multiplication* or *division* operator followed by another *factor* and so on and so forth."

What is a *factor*? For the purpose of this article a *factor* is just an integer.

Let's quickly go over the symbols used in the grammar and their meaning.

- **|** - Alternatives. A bar means "or". So (MUL | DIV) means either MUL or DIV.
- **(…)** - An open and closing parentheses mean grouping of terminals and/or non-terminals as in (MUL | DIV).
- **(…)*** - Match contents within the group zero or more times.

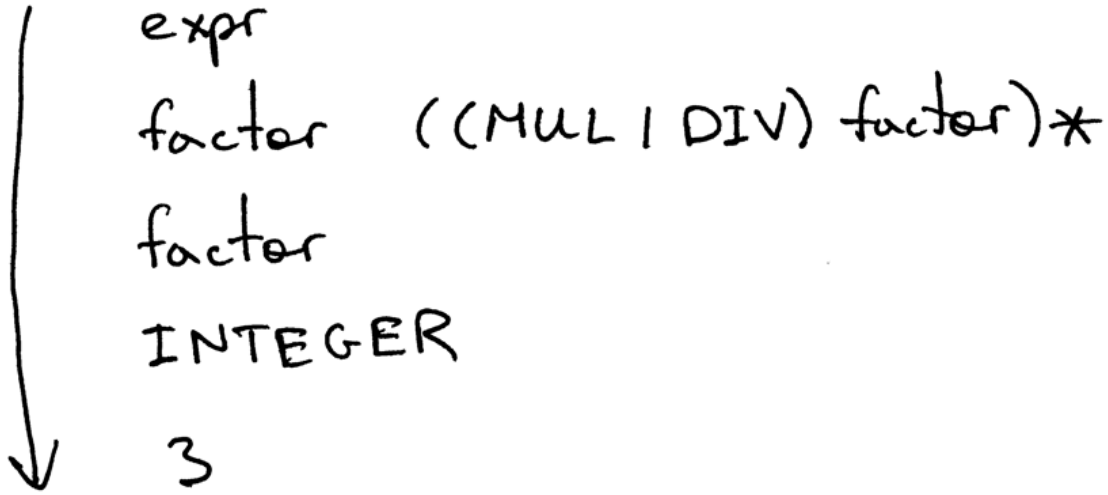
If you worked with regular expressions in the past, then the symbols **|**, **()**, and **(…)*** should be pretty familiar to you.

A grammar defines a *language* by explaining what sentences it can form. This is how you can *derive* an arithmetic expression using the grammar: first you begin with the start symbol *expr* and then repeatedly replace a non-terminal by the body of a rule for that non-terminal until you have generated a sentence

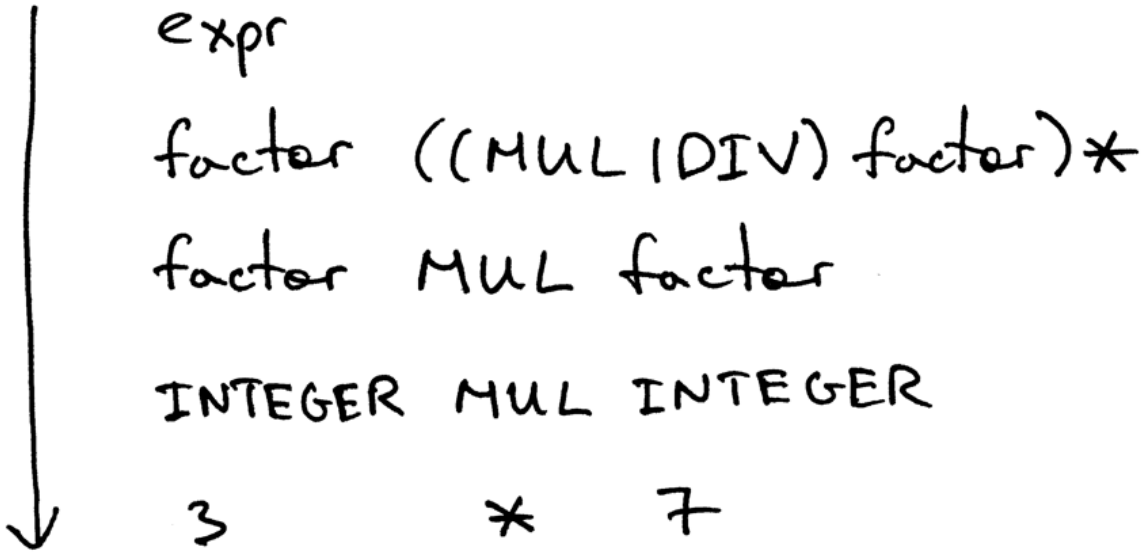
consisting solely of terminals. Those sentences form a *language* defined by the grammar.

If the grammar cannot derive a certain arithmetic expression, then it doesn't support that expression and the parser will generate a syntax error when it tries to recognize the expression.

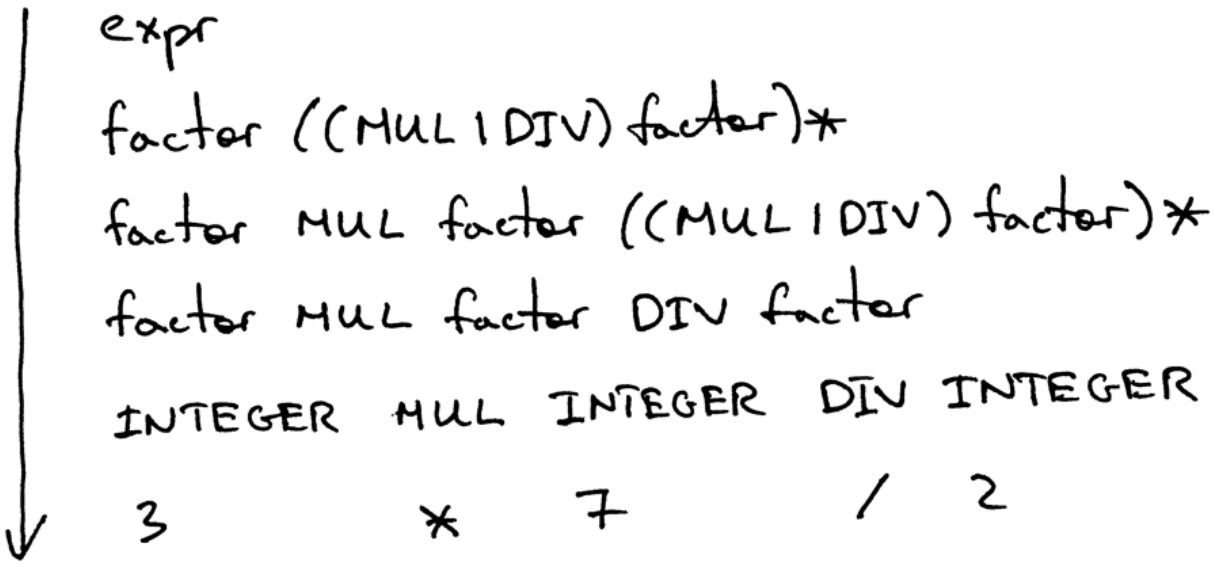
I think a couple of examples are in order. This is how the grammar derives the expression *3*:



This is how the grammar derives the expression *3 * 7*:



And this is how the grammar derives the expression *3 * 7 / 2*:



Whoa, quite a bit of theory right there!

I think when I first read about grammars, the related terminology, and all that jazz, I felt something like this:

I can assure you that I definitely was not like this:

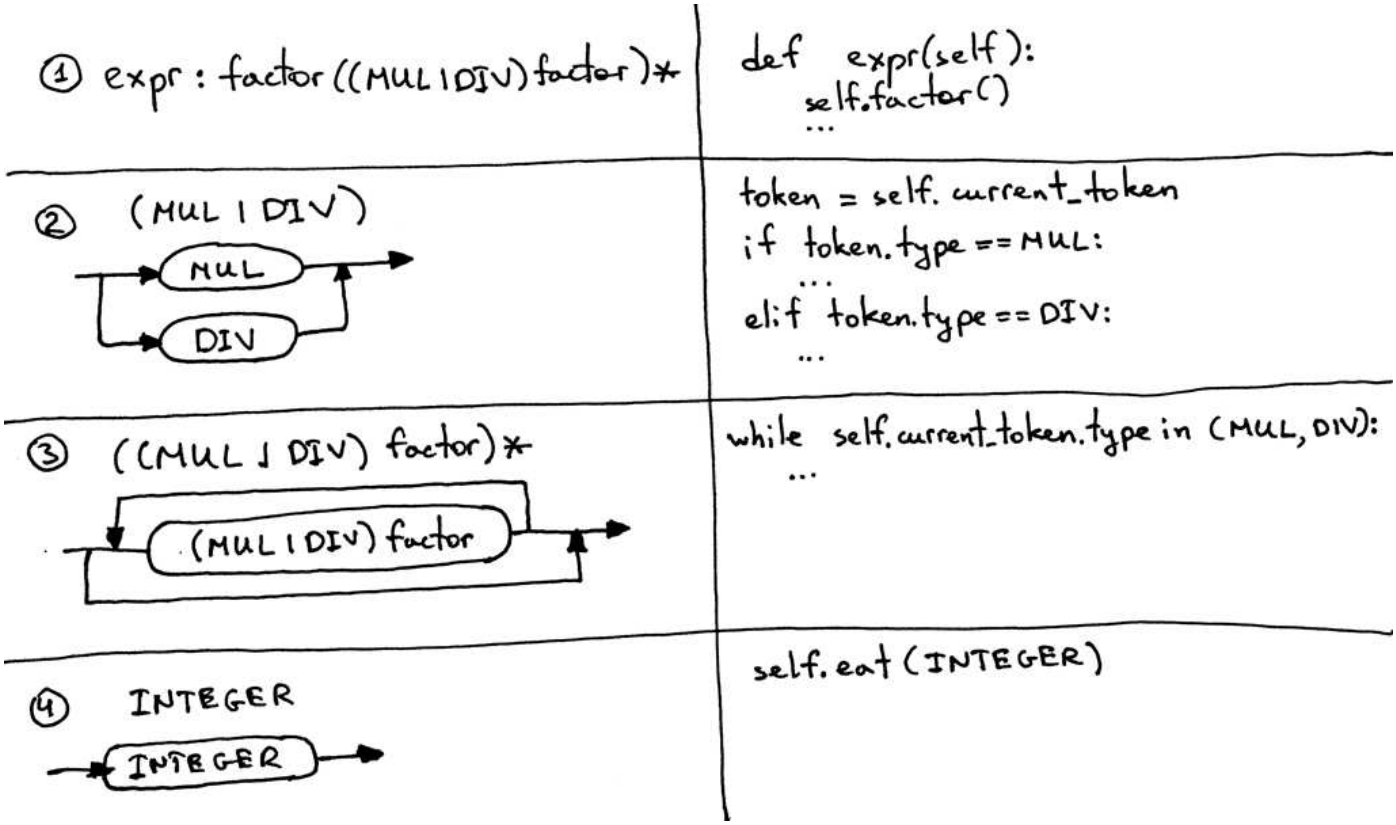
It took me some time to get comfortable with the notation, how it works, and its relationship with parsers and lexers, but I have to tell you that it pays to learn it in the long run because it's so widely used in practice and compiler literature that you're bound to run into it at some point. So, why not sooner rather than later? :)

Now, let's map that grammar to code, okay?

Here are the guidelines that we will use to convert the grammar to source code. By following them, you can literally translate the grammar to a working parser:

1. Each rule, **R**, defined in the grammar, becomes a method with the same name, and references to that rule become a method call: ***R()***. The body of the method follows the flow of the body of the rule using the very same guidelines.
2. Alternatives **(a1 | a2 | aN)** become an ***if-elif-else*** statement
3. An optional grouping **(…)*** becomes a ***while*** statement that can loop over zero or more times
4. Each token reference **T** becomes a call to the method ***eat***: ***eat(T)***. The way the *eat* method works is that it consumes the token *T* if it matches the current *lookahead* token, then it gets a new token from the lexer and assigns that token to the *current_token* internal variable.

Visually the guidelines look like this:



Let's get moving and convert our grammar to code following the above guidelines.

There are two rules in our grammar: one *expr* rule and one *factor* rule. Let's start with the *factor* rule (production). According to the guidelines, you need to create a method called *factor* (guideline 1) that has a single call to the *eat* method to consume the INTEGER token (guideline 4):

```
def factor(self):
    self.eat(INTEGER)
```

That was easy, wasn't it?

Onward!

The rule *expr* becomes the *expr* method (again according to the guideline 1). The body of the rule starts with a reference to *factor* that becomes a *factor()* method call. The optional grouping *(…)** becomes a *while* loop and *(MUL | DIV)* alternatives become an *if-elif-else* statement. By combining those pieces together we get the following *expr* method:

```
def expr(self):
    self.factor()

    while self.current_token.type in (MUL, DIV):
        token = self.current_token
        if token.type == MUL:
            self.eat(MUL)
            self.factor()
        elif token.type == DIV:
            self.eat(DIV)
            self.factor()
```

Please spend some time and study how I mapped the grammar to the source code. Make sure you understand that part because it'll come in handy later on.

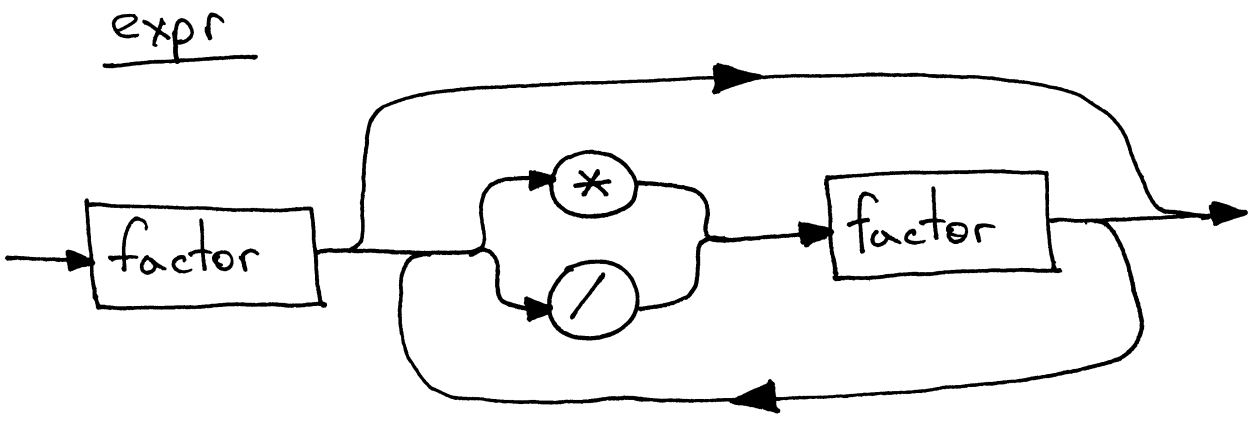
For your convenience I put the above code into the *parser.py* file that contains a lexer and a parser without an interpreter. You can download the file directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part4/parser.py) and play with it. It has an interactive prompt where you can enter expressions and see if they are valid: that is, if the parser built according to the grammar can recognize the expressions.

Here is a sample session that I ran on my computer:

```
$ python parser.py
calc> 3
calc> 3 * 7
calc> 3 * 7 / 2
calc> 3 *
Traceback (most recent call last):
  File "parser.py", line 155, in <module>
    main()
  File "parser.py", line 151, in main
    parser.parse()
  File "parser.py", line 136, in parse
    self.expr()
  File "parser.py", line 130, in expr
    self.factor()
  File "parser.py", line 114, in factor
    self.eat(INTEGER)
  File "parser.py", line 107, in eat
    self.error()
  File "parser.py", line 97, in error
    raise Exception('Invalid syntax')
Exception: Invalid syntax
```

Try it out!

I couldn't help but mention syntax diagrams again. This is how a syntax diagram for the same *expr* rule will look:



It's about time we dug into the source code of our new arithmetic expression interpreter. Below is the code of a calculator that can handle valid arithmetic expressions containing integers and any number of multiplication and division (integer division) operators. You can also see that I refactored the lexical analyzer into a separate class *Lexer* and updated the *Interpreter* class to take the *Lexer* instance as a parameter:

```
# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, MUL, DIV, EOF = 'INTEGER', 'MUL', 'DIV', 'EOF'


class Token(object):
    def __init__(self, type, value):
        # token type: INTEGER, MUL, DIV, or EOF
        self.type = type
        # token value: non-negative integer value, '*', '/', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(MUL, '*')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Lexer(object):
    def __init__(self, text):
        # client string input, e.g. "3 * 5", "12 / 3 * 4", etc
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Invalid character')

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
```

```
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '*':
                self.advance()
                return Token(MUL, '*')

            if self.current_char == '/':
                self.advance()
                return Token(DIV, '/')

            self.error()

        return Token(EOF, None)


class Interpreter(object):
    def __init__(self, lexer):
        self.lexer = lexer
        # set current token to the first token taken from the input
        self.current_token = self.lexer.get_next_token()

    def error(self):
        raise Exception('Invalid syntax')

    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.lexer.get_next_token()
        else:
            self.error()

    def factor(self):
        """Return an INTEGER token value.

        factor : INTEGER
        """
        token = self.current_token
```

```
        self.eat(INTEGER)
        return token.value

    def expr(self):
        """Arithmetic expression parser / interpreter.

        expr   : factor ((MUL | DIV) factor)*
        factor : INTEGER
        """
        result = self.factor()

        while self.current_token.type in (MUL, DIV):
            token = self.current_token
            if token.type == MUL:
                self.eat(MUL)
                result = result * self.factor()
            elif token.type == DIV:
                self.eat(DIV)
                result = result / self.factor()

        return result


def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break
        if not text:
            continue
        lexer = Lexer(text)
        interpreter = Interpreter(lexer)
        result = interpreter.expr()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into the *calc4.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part4/calc4.py). As usual, try it out and see for yourself that it works.

This is a sample session that I ran on my laptop:

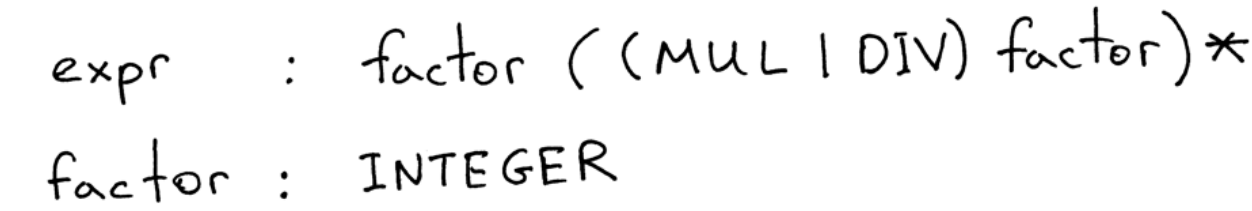
```
$ python calc4.py
calc> 7 * 4 / 2
14
calc> 7 * 4 / 2 * 3
42
calc> 10 * 4  * 2 * 3 / 8
30
```

I know you couldn't wait for this part :) Here are new exercises for today:

- Write a grammar that describes arithmetic expressions containing any number of +, -, *, or / operators. With the grammar you should be able to derive expressions like "2 + 7 * 4", "7 - 8 / 4", "14 + 2 * 3 - 6 / 2", and so on.
- Using the grammar, write an interpreter that can evaluate arithmetic expressions containing any number of +, -, *, or / operators. Your interpreter should be able to handle expressions like "2 + 7 * 4", "7 - 8 / 4", "14 + 2 * 3 - 6 / 2", and so on.
- If you've finished the above exercises, relax and enjoy :)

Check your understanding.

Keeping in mind the grammar from today's article, answer the following questions, referring to the picture below as needed:



1. What is a context-free grammar (grammar)?
2. How many rules / productions does the grammar have?
3. What is a terminal? (Identify all terminals in the picture)
4. What is a non-terminal? (Identify all non-terminals in the picture)
5. What is a head of a rule? (Identify all heads / left-hand sides in the picture)
6. What is a body of the rule? (Identify all bodies / right-hand sides in the picture)
7. What is the start symbol of a grammar?

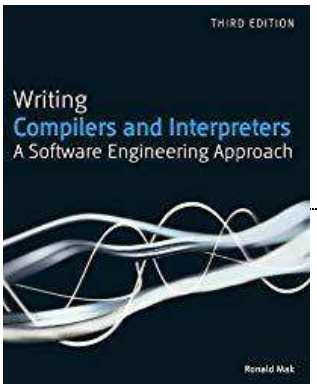
Hey, you read all the way to the end! This post contained quite a bit of theory, so I'm really proud of you that you finished it.

I'll be back next time with a new article - stay tuned and don't forget to do the exercises, they will do you good.

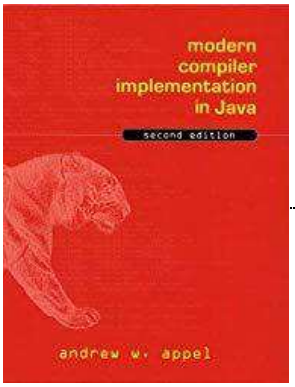
Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

   (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=FYAZBCVOB66PGR6J)
3. Modern Compiler Implementation in Java (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=ZSKKZMV7YWR22NMW)

   (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=GPMSWTZYFC2M6MJE)

Let's Build A Simple Interpreter. Part 5. (https://ruslanspivak.com/lsbasi-part5/)

Date Wed, October 14, 2015

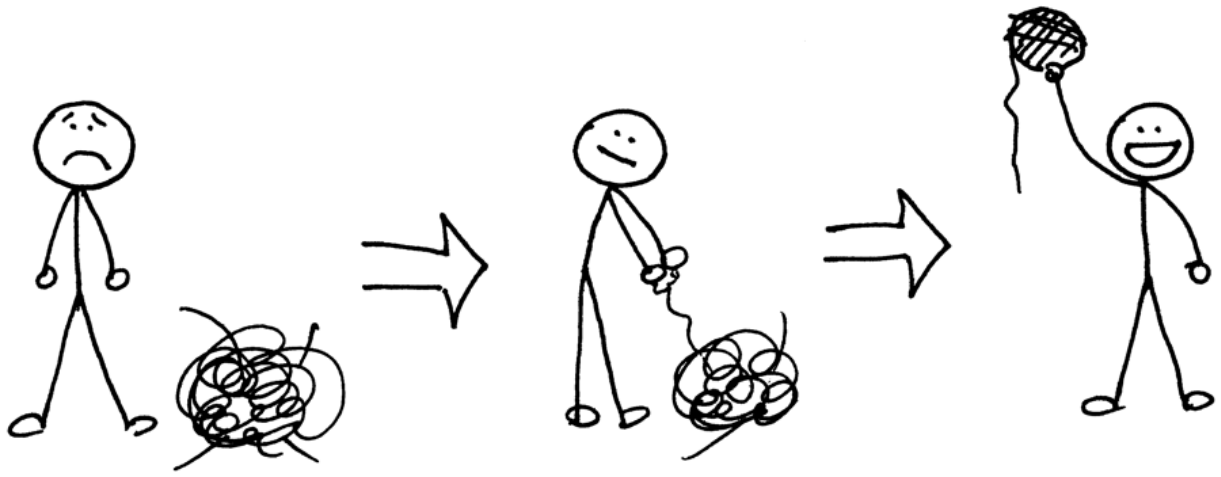
How do you tackle something as complex as understanding how to create an interpreter or compiler? In the beginning it all looks pretty much like a tangled mess of yarn that you need to untangle to get that perfect ball.

The way to get there is to just untangle it one thread, one knot at a time. Sometimes, though, you might feel like you don't understand something right away, but you have to keep going. It will eventually "click" if you're persistent enough, I promise you (Gee, if I put aside 25 cents every time I didn't understand something right away I would have become rich a long time ago :).

Probably one of the best pieces of advice I could give you on your way to understanding how to create an interpreter and compiler is to read the explanations in the articles, read the code, and then write code yourself, and even write the same code several times over a period of time to make the material and code feel natural to you, and only then move on to learn new topics. Do not rush, just slow down and take your time to understand the basic ideas deeply. This approach, while seemingly slow, will pay off down the road. Trust me.

You will eventually get your perfect ball of yarn in the end. And, you know what? Even if it is not that perfect it is still better than the alternative, which is to do nothing and not learn the topic or quickly skim over it and forget it in a couple of days.

Remember - just keep untangling: one thread, one knot at a time and practice what you've learned by writing code, a lot of it:



Today you're going to use all the knowledge you've gained from previous articles in the series and learn how to parse and interpret arithmetic expressions that have any number of addition, subtraction, multiplication, and division operators. You will write an interpreter that will be able to evaluate expressions like "14 + 2 * 3 - 6 / 2".

Before diving in and writing some code let's talk about the **associativity** and **precedence** of operators.

By convention 7 + 3 + 1 is the same as (7 + 3) + 1 and 7 - 3 - 1 is equivalent to (7 - 3) - 1. No surprises here. We all learned that at some point and have been taking it for granted since then. If we treated 7 - 3 - 1 as 7 - (3 - 1) the result would be unexpected 5 instead of the expected 3.

In ordinary arithmetic and most programming languages addition, subtraction, multiplication, and division are *left-associative*:

```
7 + 3 + 1 is equivalent to (7 + 3) + 1
7 - 3 - 1 is equivalent to (7 - 3) - 1
8 * 4 * 2 is equivalent to (8 * 4) * 2
8 / 4 / 2 is equivalent to (8 / 4) / 2
```

What does it mean for an operator to be *left-associative*?

When an operand like 3 in the expression 7 + 3 + 1 has plus signs on both sides, we need a convention to decide which operator applies to 3. Is it the one to the left or the one to the right of the operand 3? The operator + *associates* to the left because an operand that has plus signs on both sides belongs to the operator to its left and so we say that the operator + is *left-associative*. That's why 7 + 3 + 1 is equivalent to (7 + 3) + 1 by the *associativity* convention.

Okay, what about an expression like 7 + 5 * 2 where we have different kinds of operators on both sides of the operand 5? Is the expression equivalent to 7 + (5 * 2) or (7 + 5) * 2? How do we resolve this ambiguity?

In this case, the associativity convention is of no help to us because it applies only to operators of one kind, either additive (+, -) or multiplicative (*, /). We need another convention to resolve the ambiguity when we have different kinds of operators in the same expression. We need a convention that defines relative *precedence* of operators.

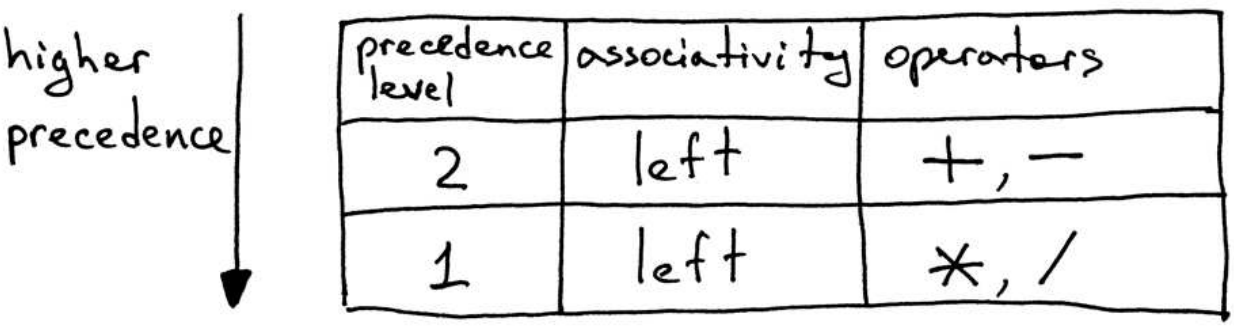
And here it is: we say that if the operator * takes its operands before + does, then it has *higher precedence*. In the arithmetic that we know and use, multiplication and division have *higher precedence* than addition and subtraction. As a result the expression 7 + 5 * 2 is equivalent to 7 + (5 * 2) and the expression 7 - 8 / 4 is equivalent to 7 - (8 / 4).

In a case where we have an expression with operators that have the same *precedence*, we just use the *associativity* convention and execute the operators from left to right:

```
7 + 3 - 1 is equivalent to (7 + 3) - 1
8 / 4 * 2 is equivalent to (8 / 4) * 2
```

I hope you didn't think I wanted to bore you to death by talking so much about the associativity and precedence of operators. The nice thing about those conventions is that we can construct a grammar for arithmetic expressions from a table that shows the associativity and precedence of arithmetic operators. Then, we can translate the grammar into code by following the guidelines I outlined in Part 4 (http://ruslanspivak.com/lsbasi-part4/), and our interpreter will be able to handle the precedence of operators in addition to associativity.

Okay, here is our precedence table:



From the table, you can tell that operators + and - have the same precedence level and they are both left-associative. You can also see that operators * and / are also left-associative, have the same precedence among themselves but have higher-precedence than addition and subtraction operators.

Here are the rules for how to construct a grammar from the precedence table:

1. For each level of precedence define a non-terminal. The body of a production for the non-terminal should contain arithmetic operators from that level and non-terminals for the next higher level of precedence.
2. Create an additional non-terminal *factor* for basic units of expression, in our case, integers. The general rule is that if you have N levels of precedence, you will need N + 1 non-terminals in total: one non-terminal for each level plus one non-terminal for basic units of expression.

Onward!

Let's follow the rules and construct our grammar.

According to Rule 1 we will define two non-terminals: a non-terminal called *expr* for level 2 and a non-terminal called *term* for level 1. And by following Rule 2 we will define a *factor* non-terminal for basic units of arithmetic expressions, integers.

The *start symbol* of our new grammar will be *expr* and the *expr* production will contain a body representing the use of operators from level 2, which in our case are operators + and - , and will contain *term* non-terminals for the next higher level of precedence, level 1:

```
expr   : term ((PLUS | MINUS) term)*
```

The *term* production will have a body representing the use of operators from level 1, which are operators * and / , and it will contain the non-terminal *factor* for the basic units of expression, integers:

```
term   : factor ((MUL | DIV) factor)*
```

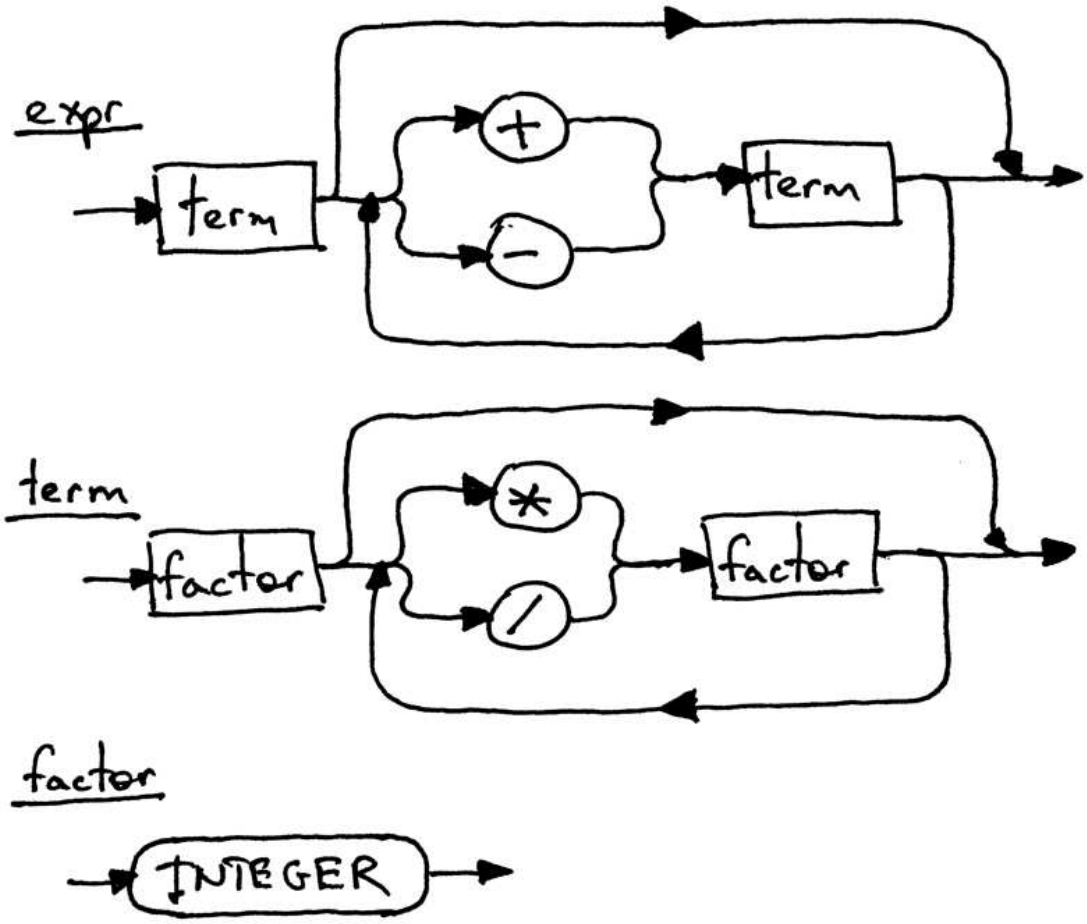
And the production for the non-terminal *factor* will be:

```
factor : INTEGER
```

You've already seen above productions as part of grammars and syntax diagrams from previous articles, but here we combine them into one grammar that takes care of the associativity and the precedence of operators:

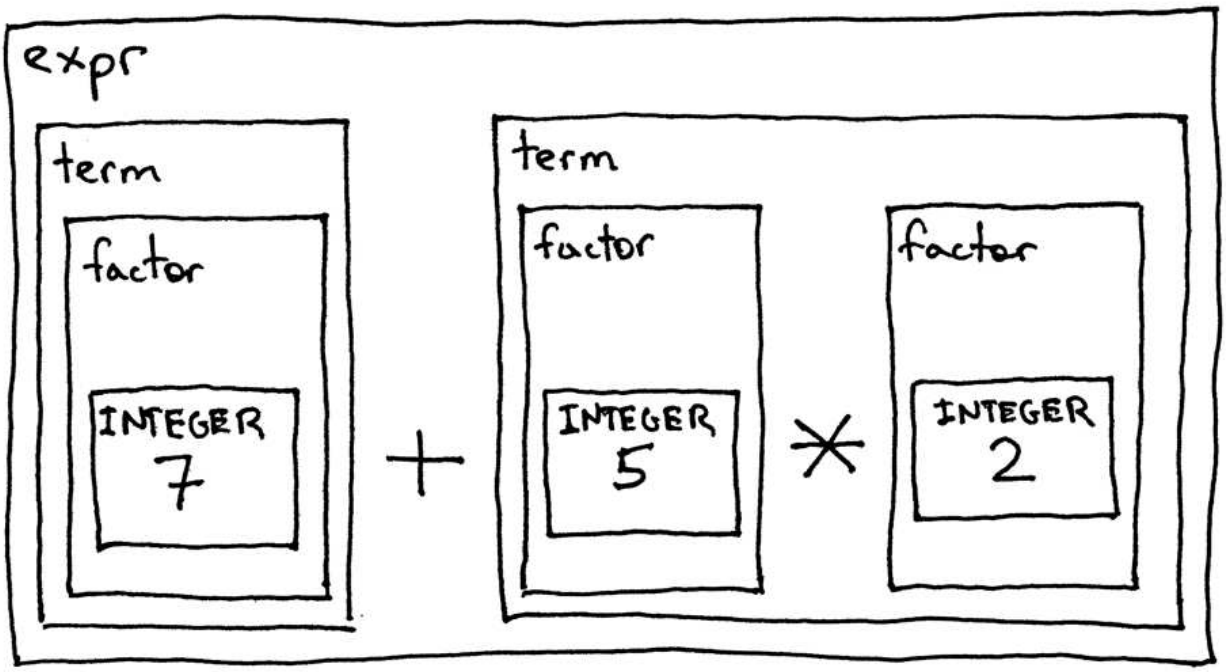
```
expr   : term ((PLUS | MINUS) term)*
term   : factor ((MUL | DIV) factor)*
factor : INTEGER
```

Here is a syntax diagram that corresponds to the grammar above:



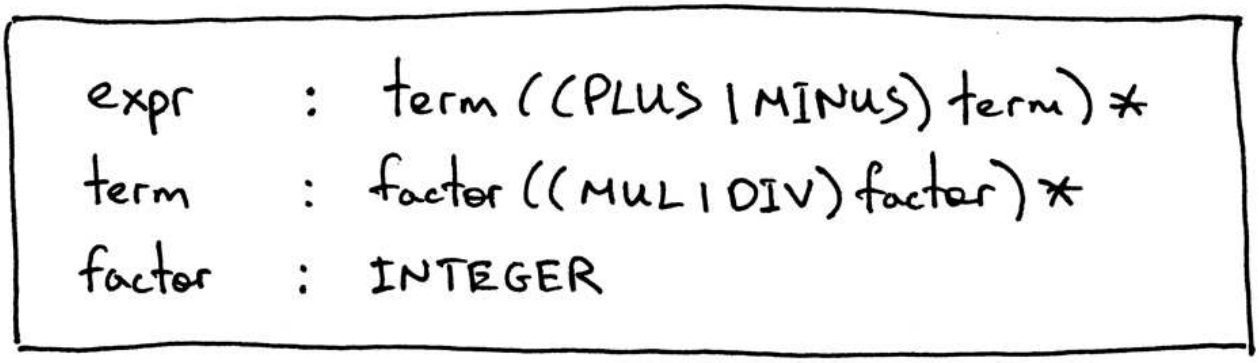
Each rectangular box in the diagram is a "method call" to another diagram. If you take the expression 7 + 5 * 2 and start with the top diagram *expr* and walk your way down to the bottommost diagram *factor*, you should be able to see that *higher-precedence* operators * and / in the lower diagram execute before operators + and - in the higher diagram.

To drive the precedence of operators point home, let's take a look at the decomposition of the same arithmetic expression 7 + 5 * 2 done in accordance with our grammar and syntax diagrams above. This is just another way to show that *higher-precedence* operators execute before operators with *lower precedence*:



Okay, let's convert the grammar to code following guidelines from Part 4 (http://ruslanspivak.com/lsbasi-part4/) and see how our new interpreter works, shall we?

Here is the grammar again:



And here is the complete code of a calculator that can handle valid arithmetic expressions containing integers and any number of addition, subtraction, multiplication, and division operators.

The following are the main changes compared with the code from Part 4 (http://ruslanspivak.com/lsbasi-part4/):

- The *Lexer* class can now tokenize +, -, *, and / (Nothing new here, we just combined code from previous articles into one class that supports all those tokens)
- Recall that each rule (production), **R**, defined in the grammar, becomes a method with the same name, and references to that rule become a method call: **R()**. As a result the *Interpreter* class now has three methods that correspond to non-terminals in the grammar: *expr*, *term*, and *factor*.

Source code:

```
# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, MINUS, MUL, DIV, EOF = (
    'INTEGER', 'PLUS', 'MINUS', 'MUL', 'DIV', 'EOF'
)


class Token(object):
    def __init__(self, type, value):
        # token type: INTEGER, PLUS, MINUS, MUL, DIV, or EOF
        self.type = type
        # token value: non-negative integer value, '+', '-', '*', '/', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS, '+')
            Token(MUL, '*')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Lexer(object):
    def __init__(self, text):
        # client string input, e.g. "3 * 5", "12 / 3 * 4", etc
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Invalid character')

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
```

```
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '+':
                self.advance()
                return Token(PLUS, '+')

            if self.current_char == '-':
                self.advance()
                return Token(MINUS, '-')

            if self.current_char == '*':
                self.advance()
                return Token(MUL, '*')

            if self.current_char == '/':
                self.advance()
                return Token(DIV, '/')

            self.error()

        return Token(EOF, None)


class Interpreter(object):
    def __init__(self, lexer):
        self.lexer = lexer
        # set current token to the first token taken from the input
        self.current_token = self.lexer.get_next_token()

    def error(self):
        raise Exception('Invalid syntax')

    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
```

```
        if self.current_token.type == token_type:
            self.current_token = self.lexer.get_next_token()
        else:
            self.error()

    def factor(self):
        """factor : INTEGER"""
        token = self.current_token
        self.eat(INTEGER)
        return token.value

    def term(self):
        """term : factor ((MUL | DIV) factor)*"""
        result = self.factor()

        while self.current_token.type in (MUL, DIV):
            token = self.current_token
            if token.type == MUL:
                self.eat(MUL)
                result = result * self.factor()
            elif token.type == DIV:
                self.eat(DIV)
                result = result / self.factor()

        return result

    def expr(self):
        """Arithmetic expression parser / interpreter.

        calc>  14 + 2 * 3 - 6 / 2
        17

        expr   : term ((PLUS | MINUS) term)*
        term   : factor ((MUL | DIV) factor)*
        factor : INTEGER
        """
        result = self.term()

        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
                result = result + self.term()
            elif token.type == MINUS:
                self.eat(MINUS)
                result = result - self.term()

        return result


def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
```

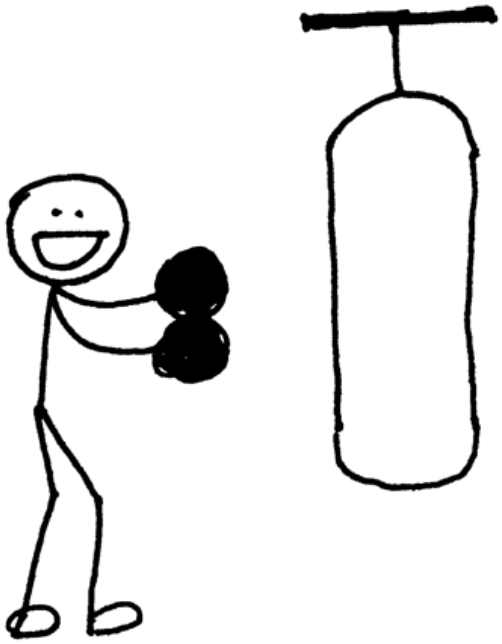
```
            break
        if not text:
            continue
        lexer = Lexer(text)
        interpreter = Interpreter(lexer)
        result = interpreter.expr()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into the *calc5.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part5/calc5.py). As usual, try it out and see for yourself that the interpreter properly evaluates arithmetic expressions that have operators with different precedence.

Here is a sample session on my laptop:

```
$ python calc5.py
calc> 3
3
calc> 2 + 7 * 4
30
calc> 7 - 8 / 4
5
calc> 14 + 2 * 3 - 6 / 2
17
```

Here are new exercises for today:



- Write an interpreter as described in this article off the top of your head, without peeking into the code from the article. Write some tests for your interpreter, and make sure they pass.
- Extend the interpreter to handle arithmetic expressions containing parentheses so that your interpreter could evaluate deeply nested arithmetic expressions like: 7 + 3 * (10 / (12 / (3 + 1) - 1))

Check your understanding.

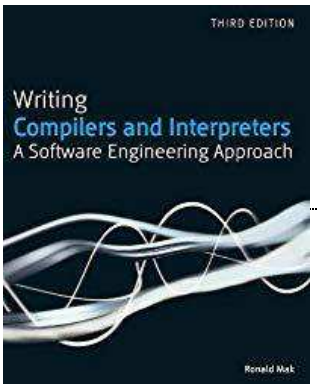
1. What does it mean for an operator to be *left-associative*?
2. Are operators + and - *left-associative* or *right-associative*? What about * and / ?
3. Does operator + have *higher precedence* than operator * ?

Hey, you read all the way to the end! That's really great. I'll be back next time with a new article - stay tuned, be brilliant, and, as usual, don't forget to do the exercises.

Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

   (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=FYAZBCVOB66PGR6J)
3. Modern Compiler Implementation in Java (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=ZSKKZMV7YWR22NMW)

Let's Build A Simple Interpreter. Part 6. (https://ruslanspivak.com/lsbasi-part6/)

Date Mon, November 02, 2015

Today is *the* day :) "Why?" you might ask. The reason is that today we're wrapping up our discussion of arithmetic expressions (well, almost) by adding parenthesized expressions to our grammar and implementing an interpreter that will be able to evaluate parenthesized expressions with arbitrarily deep nesting, like the expression 7 + 3 * (10 / (12 / (3 + 1) - 1)).

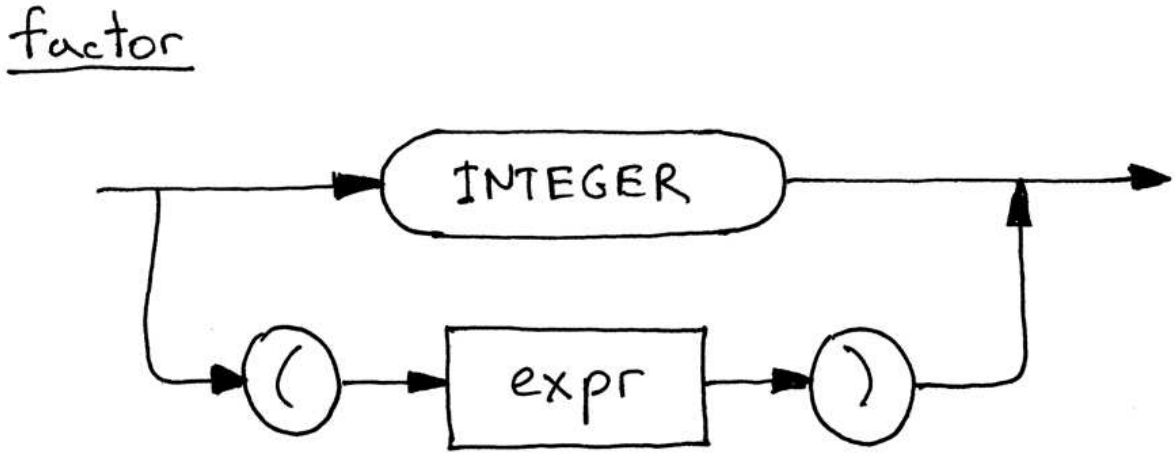
Let's get started, shall we?

First, let's modify the grammar to support expressions inside parentheses. As you remember from Part 5 (http://ruslanspivak.com/lsbasi-part5/), the *factor* rule is used for basic units in expressions. In that article, the only basic unit we had was an integer. Today we're adding another basic unit - a parenthesized expression. Let's do it.

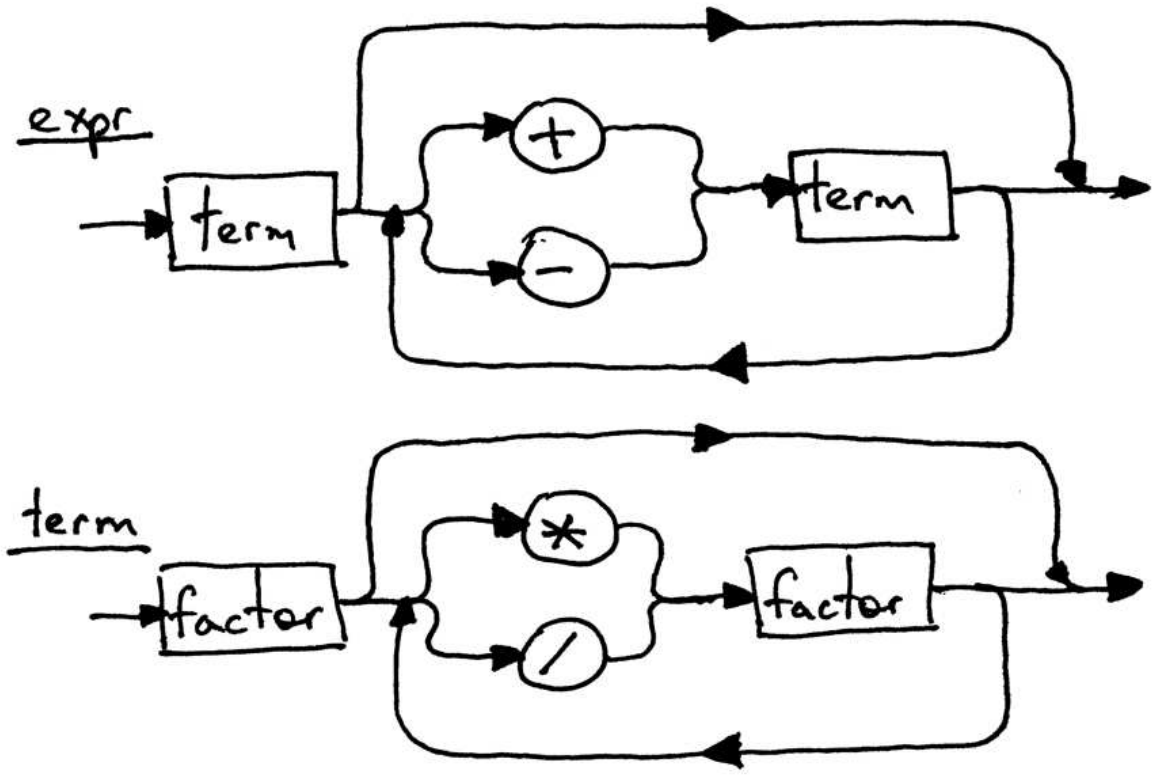
Here is our updated grammar:

The *expr* and the *term* productions are exactly the same as in Part 5 (http://ruslanspivak.com/lsbasi-part5/) and the only change is in the *factor* production where the terminal LPAREN represents a left parenthesis '(', the terminal RPAREN represents a right parenthesis ')', and the non-terminal *expr* between the parentheses refers to the *expr* rule.

Here is the updated syntax diagram for the *factor*, which now includes alternatives:

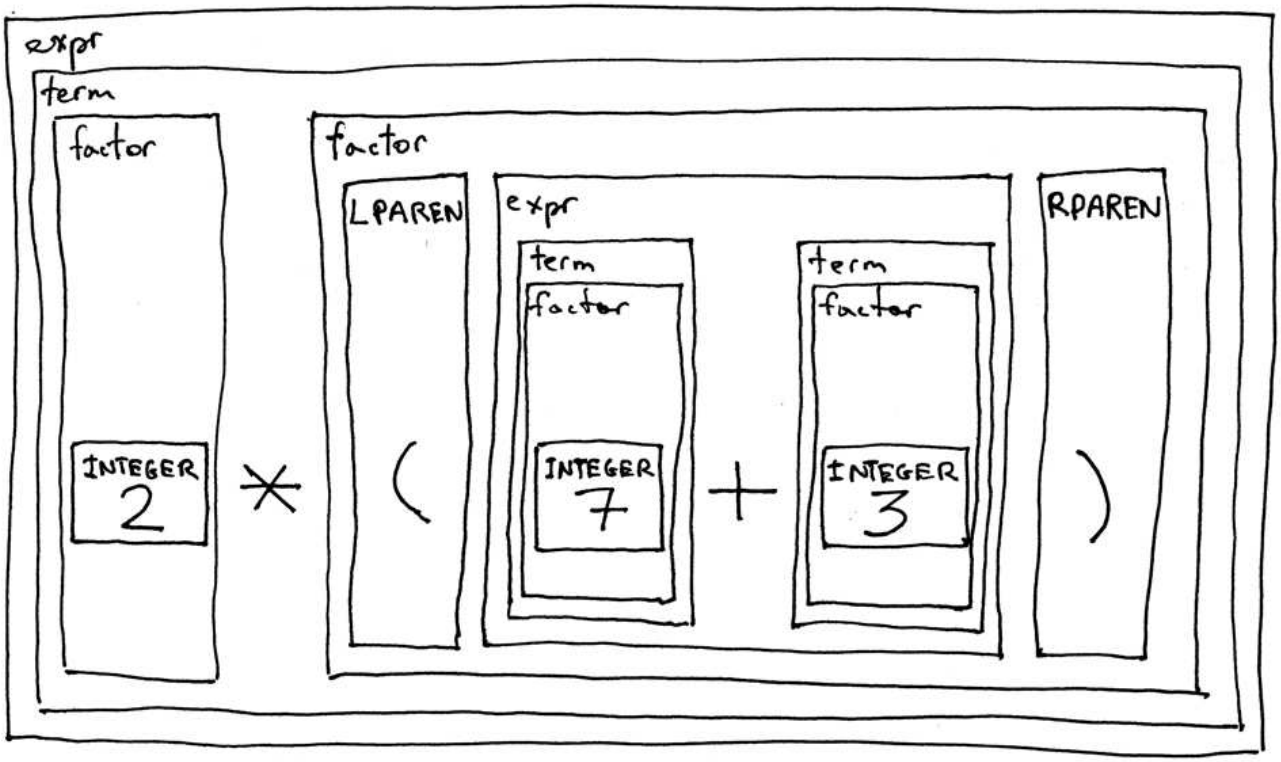


Because the grammar rules for the *expr* and the *term* haven't changed, their syntax diagrams look the same as in Part 5 (http://ruslanspivak.com/lsbasi-part5/):



Here is an interesting feature of our new grammar - it is recursive. If you try to derive the expression 2 * (7 + 3), you will start with the *expr* start symbol and eventually you will get to a point where you will recursively use the *expr* rule again to derive the (7 + 3) portion of the original arithmetic expression.

Let's decompose the expression 2 * (7 + 3) according to the grammar and see how it looks:



A little aside: if you need a refresher on recursion, take a look at Daniel P. Friedman and Matthias Felleisen's The Little Schemer (http://www.amazon.com/gp/product/0262560992/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0262560992&linkCode=as2&tag=russblo0b-20&linkId=IM7CT7RLWNGJ7J54) book - it's really good.

Okay, let's get moving and translate our new updated grammar to code.

The following are the main changes to the code from the previous article:

1. The *Lexer* has been modified to return two more tokens: LPAREN for a left parenthesis and RPAREN for a right parenthesis.
2. The *Interpreter*'s *factor* method has been slightly updated to parse parenthesized expressions in addition to integers.

Here is the complete code of a calculator that can evaluate arithmetic expressions containing integers; any number of addition, subtraction, multiplication and division operators; and parenthesized expressions with arbitrarily deep nesting:

```
# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, MINUS, MUL, DIV, LPAREN, RPAREN, EOF = (
    'INTEGER', 'PLUS', 'MINUS', 'MUL', 'DIV', '(', ')', 'EOF'
)


class Token(object):
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS, '+')
            Token(MUL, '*')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Lexer(object):
    def __init__(self, text):
        # client string input, e.g. "4 + 2 * 3 - 6 / 2"
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Invalid character')

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
```

```
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '+':
                self.advance()
                return Token(PLUS, '+')

            if self.current_char == '-':
                self.advance()
                return Token(MINUS, '-')

            if self.current_char == '*':
                self.advance()
                return Token(MUL, '*')

            if self.current_char == '/':
                self.advance()
                return Token(DIV, '/')

            if self.current_char == '(':
                self.advance()
                return Token(LPAREN, '(')

            if self.current_char == ')':
                self.advance()
                return Token(RPAREN, ')')

            self.error()

        return Token(EOF, None)


class Interpreter(object):
    def __init__(self, lexer):
        self.lexer = lexer
        # set current token to the first token taken from the input
        self.current_token = self.lexer.get_next_token()

    def error(self):
        raise Exception('Invalid syntax')
```

```
    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.lexer.get_next_token()
        else:
            self.error()

    def factor(self):
        """factor : INTEGER | LPAREN expr RPAREN"""
        token = self.current_token
        if token.type == INTEGER:
            self.eat(INTEGER)
            return token.value
        elif token.type == LPAREN:
            self.eat(LPAREN)
            result = self.expr()
            self.eat(RPAREN)
            return result

    def term(self):
        """term : factor ((MUL | DIV) factor)*"""
        result = self.factor()

        while self.current_token.type in (MUL, DIV):
            token = self.current_token
            if token.type == MUL:
                self.eat(MUL)
                result = result * self.factor()
            elif token.type == DIV:
                self.eat(DIV)
                result = result / self.factor()

        return result

    def expr(self):
        """Arithmetic expression parser / interpreter.

        calc> 7 + 3 * (10 / (12 / (3 + 1) - 1))
        22

        expr   : term ((PLUS | MINUS) term)*
        term   : factor ((MUL | DIV) factor)*
        factor : INTEGER | LPAREN expr RPAREN
        """
        result = self.term()

        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
                result = result + self.term()
            elif token.type == MINUS:
                self.eat(MINUS)
```

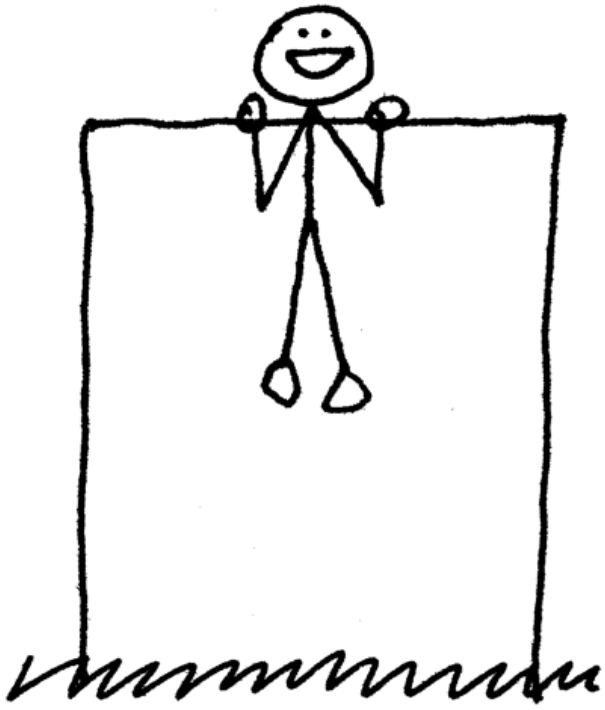
```
                result = result - self.term()

        return result


def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break
        if not text:
            continue
        lexer = Lexer(text)
        interpreter = Interpreter(lexer)
        result = interpreter.expr()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into the calc6.py (https://github.com/rspivak/lsbasi/blob/master/part6/calc6.py) file, try it out and see for yourself that your new interpreter properly evaluates arithmetic expressions that have different operators and parentheses.

Here is a sample session:

```
$ python calc6.py
calc> 3
3
calc> 2 + 7 * 4
30
calc> 7 - 8 / 4
5
calc> 14 + 2 * 3 - 6 / 2
17
calc> 7 + 3 * (10 / (12 / (3 + 1) - 1))
22
calc> 7 + 3 * (10 / (12 / (3 + 1) - 1)) / (2 + 3) - 5 - 3 + (8)
10
calc> 7 + (((3 + 2)))
12
```

And here is a new exercise for you for today:



- Write your own version of the interpreter of arithmetic expressions as described in this article. Remember: repetition is the mother of all learning.

Hey, you read all the way to the end! Congratulations, you've just learned how to create (and if you've done the exercise - you've actually written) a basic *recursive-descent parser / interpreter* that can evaluate pretty complex arithmetic expressions.

In the next article I will talk in a lot more detail about *recursive-descent parsers*. I will also introduce an important and widely used data structure in interpreter and compiler construction that we'll use throughout the series.

Stay tuned and see you soon. Until then, keep working on your interpreter and most importantly: have fun and enjoy the process!

Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

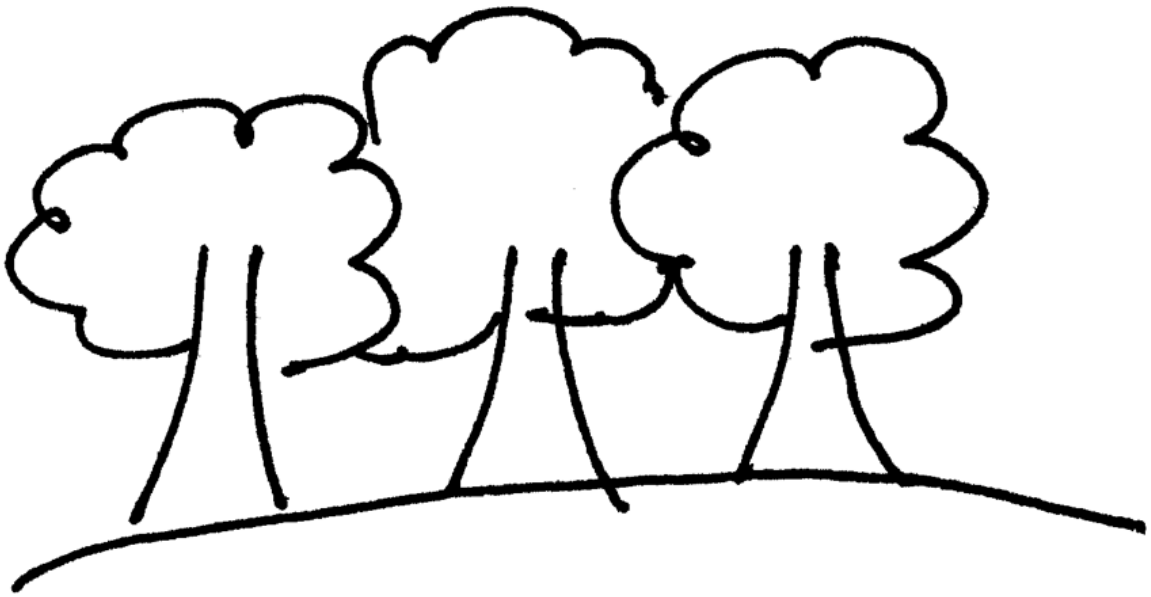
Let's Build A Simple Interpreter. Part 7. (https://ruslanspivak.com/lsbasi-part7/)

Date Tue, December 15, 2015

As I promised you last time, today I will talk about one of the central data structures that we'll use throughout the rest of the series, so buckle up and let's go.

Up until now, we had our interpreter and parser code mixed together and the interpreter would evaluate an expression as soon as the parser recognized a certain language construct like addition, subtraction, multiplication, or division. Such interpreters are called *syntax-directed interpreters*. They usually make a single pass over the input and are suitable for basic language applications. In order to analyze more complex Pascal programming language constructs, we need to build an *intermediate representation* (*IR*). Our parser will be responsible for building an *IR* and our interpreter will use it to interpret the input represented as the *IR*.

It turns out that a tree is a very suitable data structure for an IR.

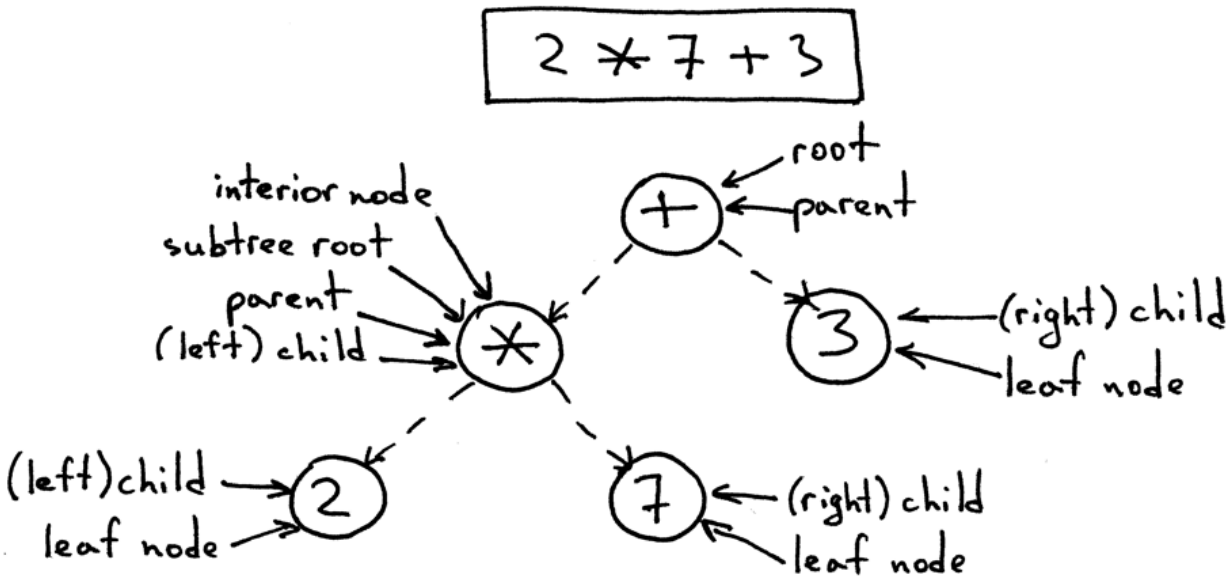


Let's quickly talk about tree terminology.

- A *tree* is a data structure that consists of one or more nodes organized into a hierarchy.
- The tree has one *root*, which is the top node.
- All nodes except the root have a unique *parent*.
- The node labeled ***** in the picture below is a *parent*. Nodes labeled **2** and **7** are its *children*; children are ordered from left to right.

- A node with no children is called a *leaf* node.
- A node that has one or more children and that is not the root is called an *interior* node.
- The children can also be complete *subtrees*. In the picture below the left child (labeled *****) of the **+** node is a complete *subtree* with its own children.
- In computer science we draw trees upside down starting with the root node at the top and branches growing downward.

Here is a tree for the expression 2 * 7 + 3 with explanations:

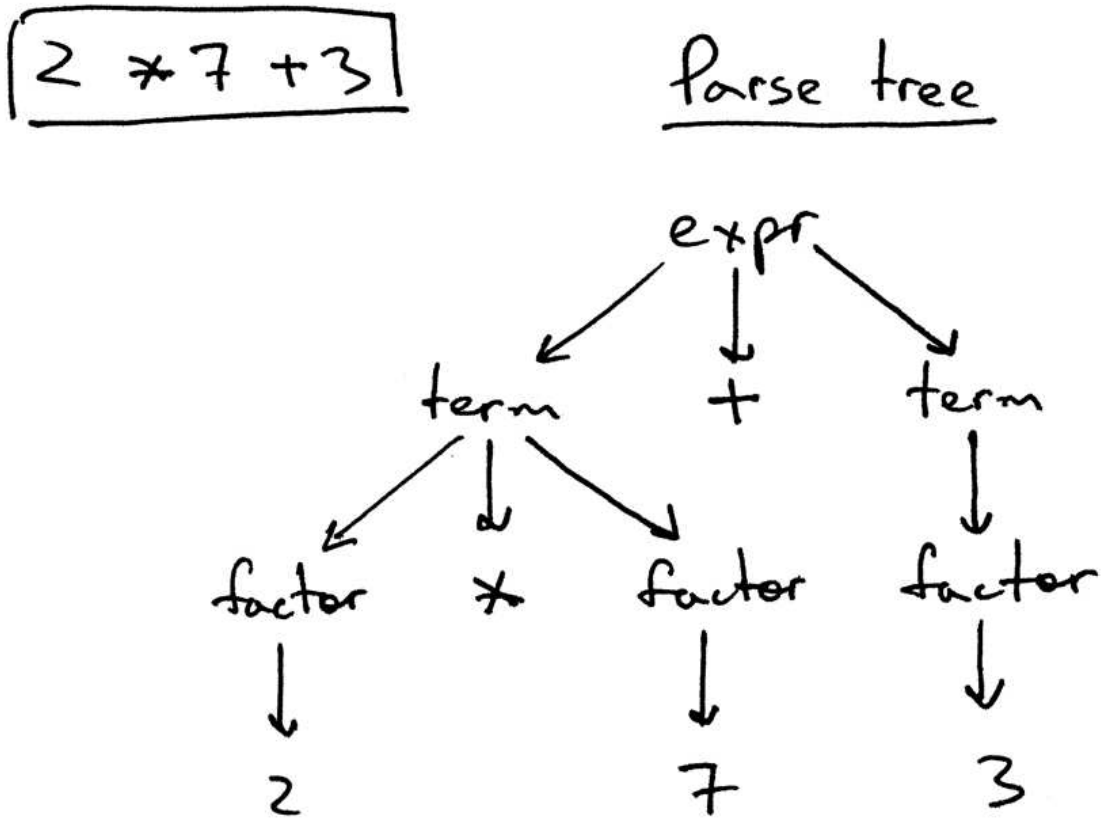


The IR we'll use throughout the series is called an *abstract-syntax tree* (*AST*). But before we dig deeper into ASTs let's talk about *parse trees* briefly. Though we're not going to use parse trees for our interpreter and compiler, they can help you understand how your parser interpreted the input by visualizing the execution trace of the parser. We'll also compare them with ASTs to see why ASTs are better suited for intermediate representation than parse trees.

So, what is a parse tree? A *parse-tree* (sometimes called a *concrete syntax tree*) is a tree that represents the syntactic structure of a language construct according to our grammar definition. It basically shows how your parser recognized the language construct or, in other words, it shows how the start symbol of your grammar derives a certain string in the programming language.

The call stack of the parser implicitly represents a parse tree and it's automatically built in memory by your parser as it is trying to recognize a certain language construct.

Let's take a look at a parse tree for the expression 2 * 7 + 3:



In the picture above you can see that:

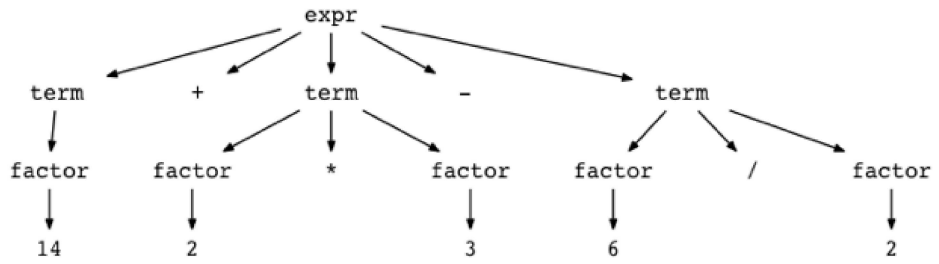
- The parse tree records a sequence of rules the parser applies to recognize the input.
- The root of the parse tree is labeled with the grammar start symbol.
- Each interior node represents a non-terminal, that is it represents a grammar rule application, like *expr*, *term*, or *factor* in our case.
- Each leaf node represents a token.

As I've already mentioned, we're not going to manually construct parser trees and use them for our interpreter but parse trees can help you understand how the parser interpreted the input by visualizing the parser call sequence.

You can see how parse trees look like for different arithmetic expressions by trying out a small utility called genptdot.py (https://github.com/rspivak/lsbasi/blob/master/part7/python/genptdot.py) that I quickly wrote to help you visualize them. To use the utility you first need to install Graphviz (http://graphviz.org) package and after you've run the following command, you can open the generated image file parsetree.png and see a parse tree for the expression you passed as a command line argument:

```
$ python genptdot.py "14 + 2 * 3 - 6 / 2" > \
  parsetree.dot && dot -Tpng -o parsetree.png parsetree.dot
```

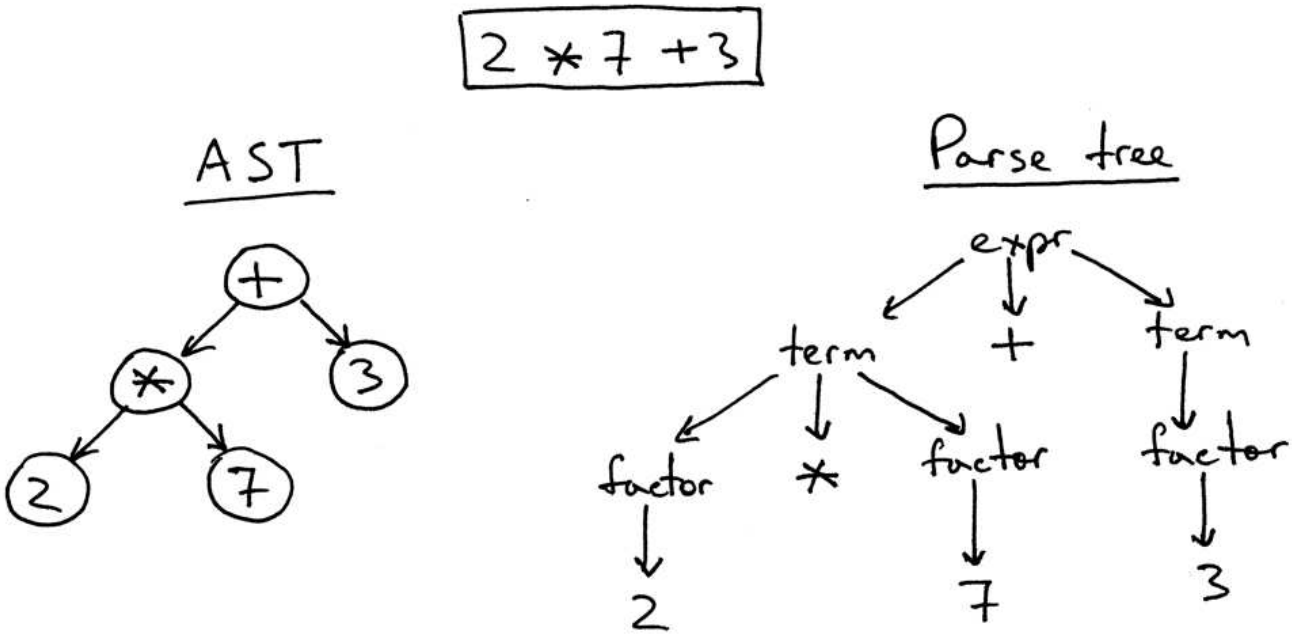
Here is the generated image parsetree.png for the expression 14 + 2 * 3 - 6 / 2:



Play with the utility a bit by passing it different arithmetic expressions and see what a parse tree looks like for a particular expression.

Now, let's talk about *abstract-syntax trees* (AST). This is the *intermediate representation* (IR) that we'll heavily use throughout the rest of the series. It is one of the central data structures for our interpreter and future compiler projects.

Let's start our discussion by taking a look at both the AST and the parse tree for the expression 2 * 7 + 3:



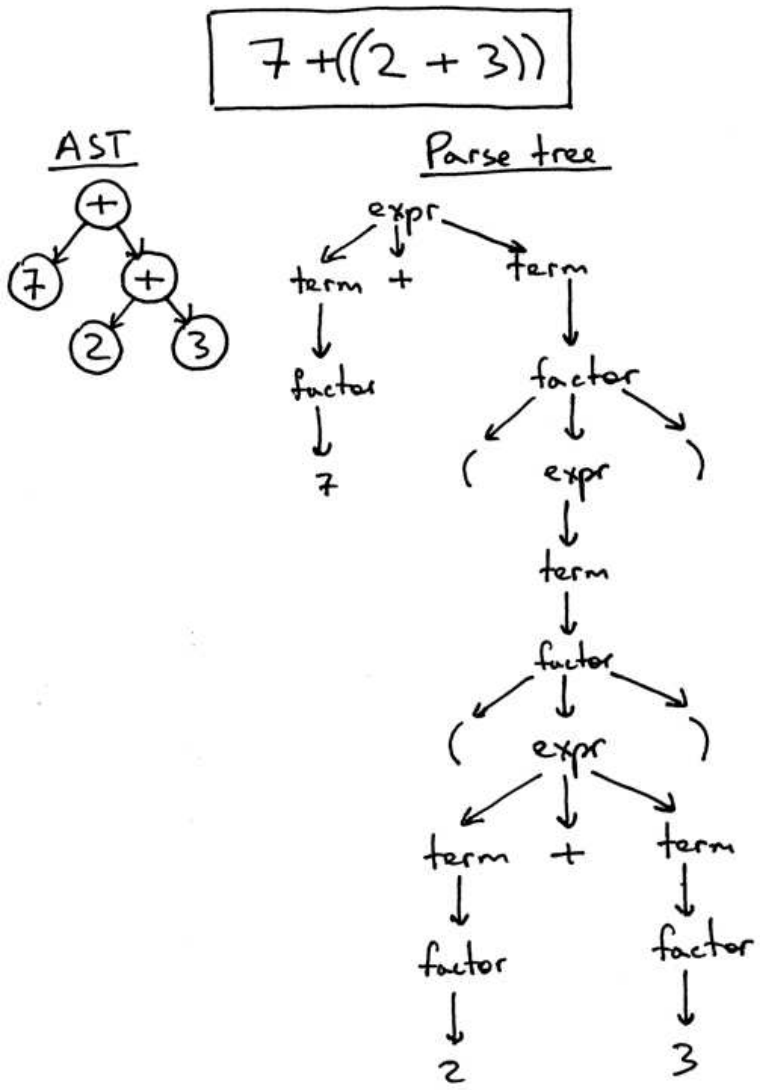
As you can see from the picture above, the AST captures the essence of the input while being smaller.

Here are the main differences between ASTs and Parse trees:

- ASTs uses operators/operations as root and interior nodes and it uses operands as their children.
- ASTs do not use interior nodes to represent a grammar rule, unlike the parse tree does.
- ASTs don't represent every detail from the real syntax (that's why they're called *abstract*) - no rule nodes and no parentheses, for example.
- ASTs are dense compared to a parse tree for the same language construct.

So, what is an abstract syntax tree? An *abstract syntax tree* (*AST*) is a tree that represents the abstract syntactic structure of a language construct where each interior node and the root node represents an operator, and the children of the node represent the operands of that operator.

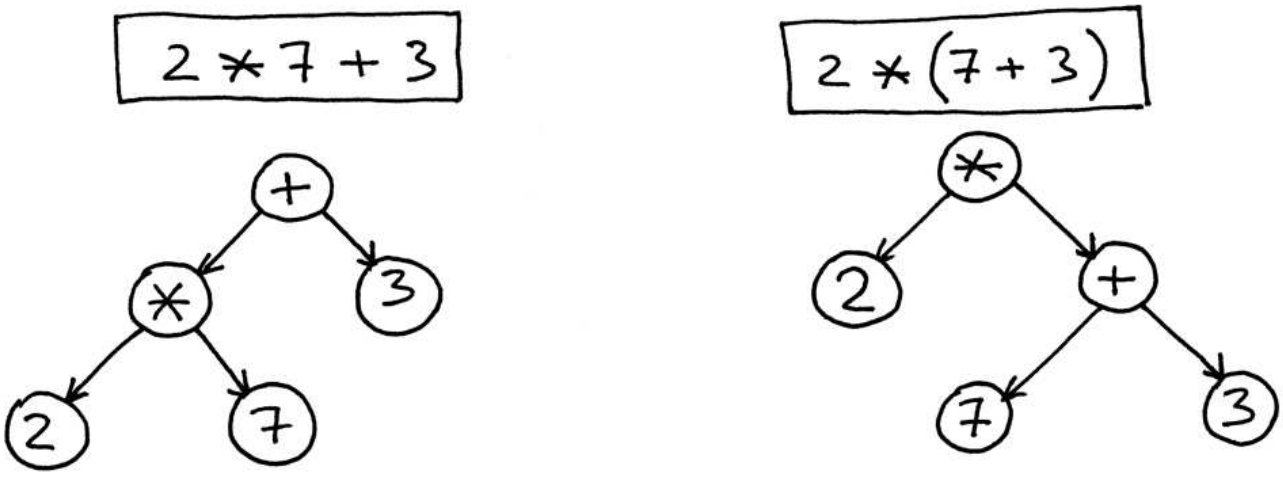
I've already mentioned that ASTs are more compact than parse trees. Let's take a look at an AST and a parse tree for the expression 7 + ((2 + 3)). You can see that the following AST is much smaller than the parse tree, but still captures the essence of the input:



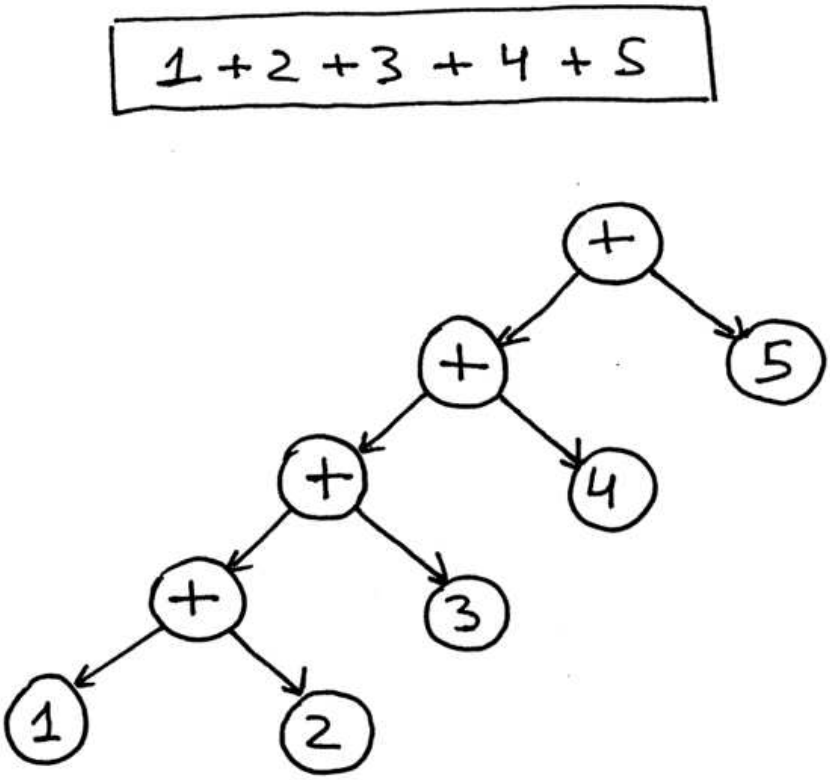
So far so good, but how do you encode operator precedence in an AST? In order to encode the operator precedence in AST, that is, to represent that "X happens before Y" you just need to put X lower in the tree than Y. And you've already seen that in the previous pictures.

Let's take a look at some more examples.

In the picture below, on the left, you can see an AST for the expression 2 * 7 + 3. Let's change the precedence by putting 7 + 3 inside the parentheses. You can see, on the right, what an AST looks like for the modified expression 2 * (7 + 3):



Here is an AST for the expression 1 + 2 + 3 + 4 + 5:



From the pictures above you can see that operators with higher precedence end up being lower in the tree.

Okay, let's write some code to implement different AST node types and modify our parser to generate an AST tree composed of those nodes.

First, we'll create a base node class called AST that other classes will inherit from:

```
class AST(object):
    pass
```

Not much there, actually. Recall that ASTs represent the operator-operand model. So far, we have four operators and integer operands. The operators are addition, subtraction, multiplication, and division. We could have created a separate class to represent each operator like AddNode, SubNode, MulNode, and DivNode, but instead we're going to have only one *BinOp* class to represent all four binary operators (a *binary operator* is an operator that operates on two operands):

```
class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right
```

The parameters to the constructor are *left*, *op*, and *right*, where *left* and *right* point correspondingly to the node of the left operand and to the node of the right operand. *Op* holds a token for the operator itself: Token(PLUS, '+') for the plus operator, Token(MINUS, '-') for the minus operator, and so on.

To represent integers in our AST, we'll define a class *Num* that will hold an INTEGER token and the token's value:

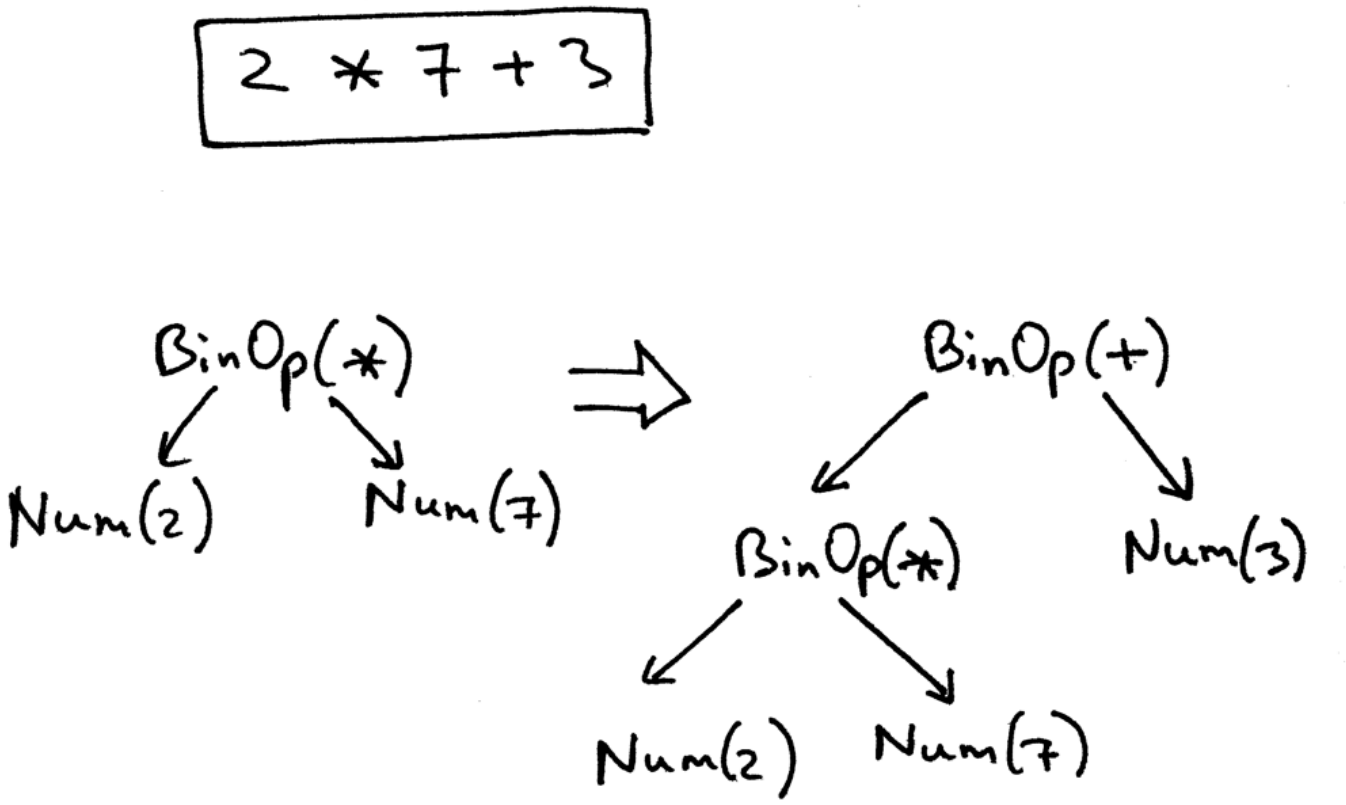
```
class Num(AST):
    def __init__(self, token):
        self.token = token
        self.value = token.value
```

As you've noticed, all nodes store the token used to create the node. This is mostly for convenience and it will come in handy in the future.

Recall the AST for the expression 2 * 7 + 3. We're going to manually create it in code for that expression:

```
>>> from spi import Token, MUL, PLUS, INTEGER, Num, BinOp
>>>
>>> mul_token = Token(MUL, '*')
>>> plus_token = Token(PLUS, '+')
>>> mul_node = BinOp(
...     left=Num(Token(INTEGER, 2)),
...     op=mul_token,
...     right=Num(Token(INTEGER, 7))
... )
>>> add_node = BinOp(
...     left=mul_node,
...     op=plus_token,
...     right=Num(Token(INTEGER, 3))
... )
```

Here is how an AST will look with our new node classes defined. The picture below also follows the manual construction process above:



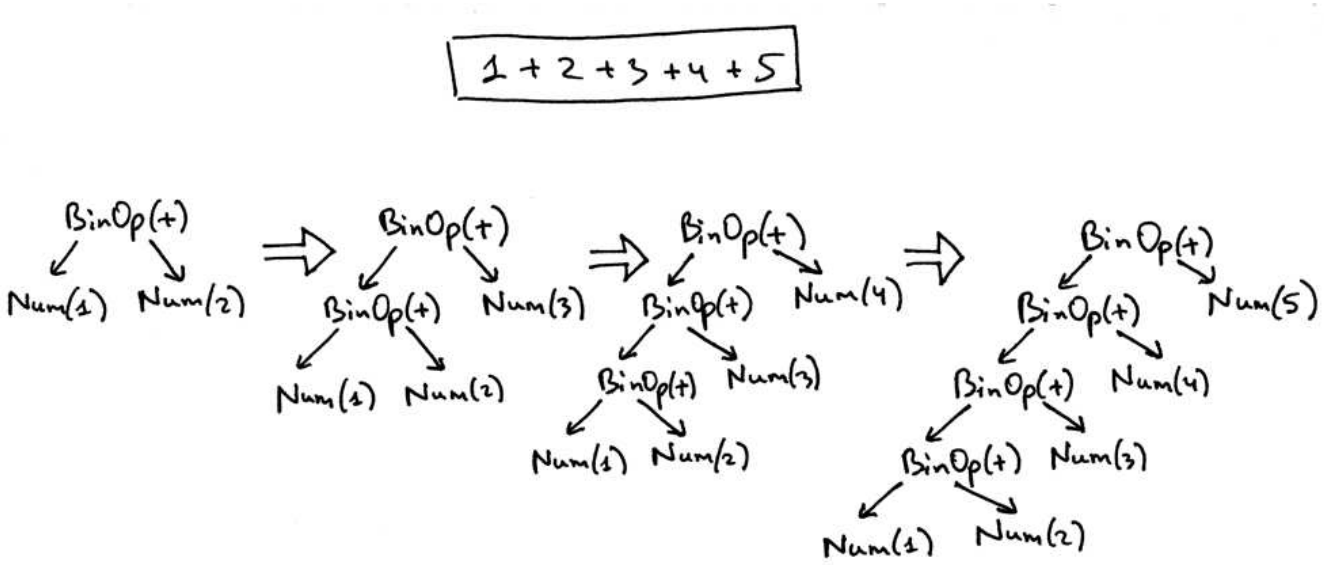
Here is our modified parser code that builds and returns an AST as a result of recognizing the input (an arithmetic expression):

```
class AST(object):
    pass


class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right


class Num(AST):
    def __init__(self, token):
        self.token = token
        self.value = token.value


class Parser(object):
    def __init__(self, lexer):
        self.lexer = lexer
        # set current token to the first token taken from the input
        self.current_token = self.lexer.get_next_token()

    def error(self):
        raise Exception('Invalid syntax')

    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.lexer.get_next_token()
        else:
            self.error()

    def factor(self):
        """factor : INTEGER | LPAREN expr RPAREN"""
        token = self.current_token
        if token.type == INTEGER:
            self.eat(INTEGER)
            return Num(token)
        elif token.type == LPAREN:
            self.eat(LPAREN)
            node = self.expr()
            self.eat(RPAREN)
            return node

    def term(self):
        """term : factor ((MUL | DIV) factor)*"""
        node = self.factor()

        while self.current_token.type in (MUL, DIV):
            token = self.current_token
            if token.type == MUL:
                self.eat(MUL)
            elif token.type == DIV:
```

```
                self.eat(DIV)

            node = BinOp(left=node, op=token, right=self.factor())

        return node

    def expr(self):
        """
        expr   : term ((PLUS | MINUS) term)*
        term   : factor ((MUL | DIV) factor)*
        factor : INTEGER | LPAREN expr RPAREN
        """
        node = self.term()

        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
            elif token.type == MINUS:
                self.eat(MINUS)

            node = BinOp(left=node, op=token, right=self.term())

        return node

    def parse(self):
        return self.expr()
```

Let's go over the process of an AST construction for some arithmetic expressions.

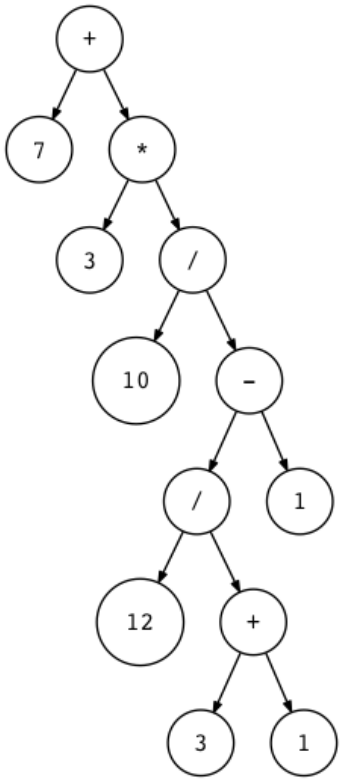
If you look at the parser code above you can see that the way it builds nodes of an AST is that each BinOp node adopts the current value of the *node* variable as its left child and the result of a call to a *term* or *factor* as its right child, so it's effectively pushing down nodes to the left and the tree for the expression 1 +2 + 3 + 4 + 5 below is a good example of that. Here is a visual representation how the parser gradually builds an AST for the expression 1 + 2 + 3 + 4 + 5:



To help you visualize ASTs for different arithmetic expressions, I wrote a small utility that takes an arithmetic expression as its first argument and generates a DOT file that is then processed by the *dot* utility to actually draw an AST for you (*dot* is part of the Graphviz (http://graphviz.org) package that you

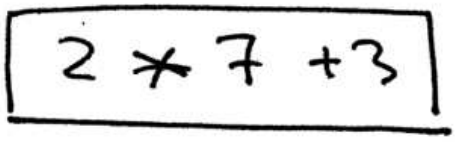
need to install to run the *dot* command). Here is a command and a generated AST image for the expression 7 + 3 * (10 / (12 / (3 + 1) - 1)):

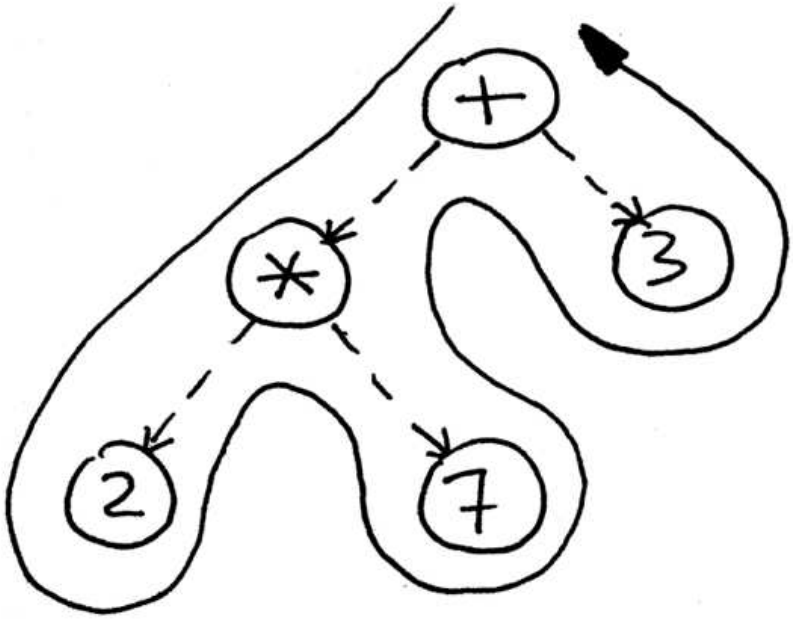
```
$ python genastdot.py "7 + 3 * (10 / (12 / (3 + 1) - 1))" > \
  ast.dot && dot -Tpng -o ast.png ast.dot
```



It's worth your while to write some arithmetic expressions, manually draw ASTs for the expressions, and then verify them by generating AST images for the same expressions with the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part7/python/genastdot.py) tool. That will help you better understand how ASTs are constructed by the parser for different arithmetic expressions.

Okay, here is an AST for the expression 2 * 7 + 3:



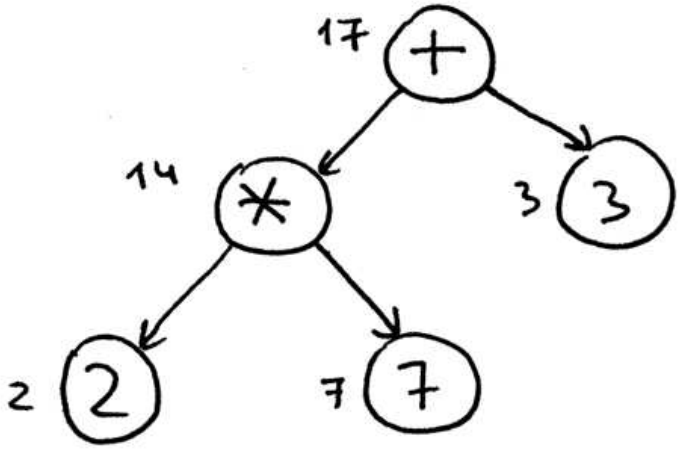


How do you navigate the tree to properly evaluate the expression represented by that tree? You do that by using a *postorder traversal* - a special case of *depth-first traversal* - which starts at the root node and recursively visits the children of each node from left to right. The postorder traversal visits nodes as far away from the root as fast as it can.

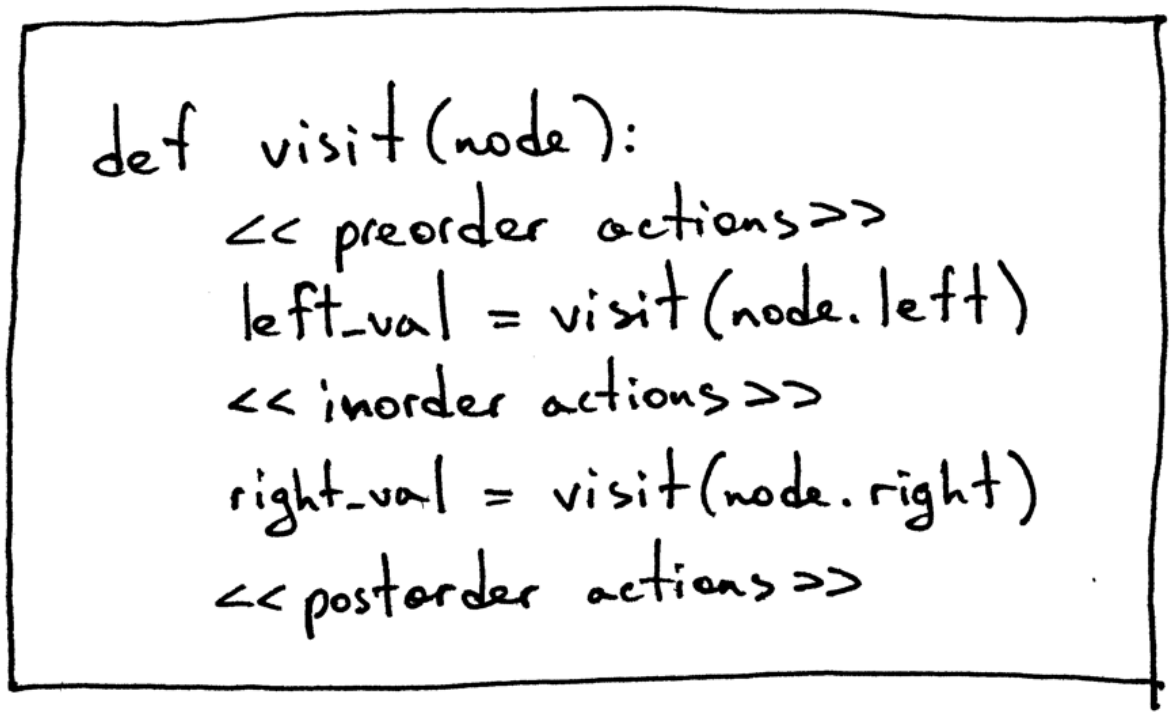
Here is a pseudo code for the postorder traversal where <<*postorder actions*>> is a placeholder for actions like addition, subtraction, multiplication, or division for a *BinOp* node or a simpler action like returning the integer value of a *Num* node:

```
def visit(node):
    # for every child node from left to right
    for child in node.children:
        visit(child)
    << postorder actions >>
```

The reason we're going to use a postorder traversal for our interpreter is that first, we need to evaluate interior nodes lower in the tree because they represent operators with higher precedence and second, we need to evaluate operands of an operator before applying the operator to those operands. In the picture below, you can see that with postorder traversal we first evaluate the expression 2 * 7 and only after that we evaluate 14 + 3, which gives us the correct result, 17:



For the sake of completeness, I'll mention that there are three types of depth-first traversal: *preorder traversal*, *inorder traversal*, and *postorder traversal*. The name of the traversal method comes from the place where you put actions in the visitation code:



Sometimes you might have to execute certain actions at all those points (preorder, inorder, and postorder). You'll see some examples of that in the source code repository for this article.

Okay, let's write some code to visit and interpret the abstract syntax trees built by our parser, shall we?

Here is the source code that implements the Visitor pattern (https://en.wikipedia.org/wiki/Visitor_pattern):

```
class NodeVisitor(object):
    def visit(self, node):
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception('No visit_{} method'.format(type(node).__name__))
```

And here is the source code of our *Interpreter* class that inherits from the *NodeVisitor* class and implements different methods that have the form *visit_NodeType*, where *NodeType* is replaced with the node's class name like *BinOp*, *Num* and so on:

```
class Interpreter(NodeVisitor):
    def __init__(self, parser):
        self.parser = parser

    def visit_BinOp(self, node):
        if node.op.type == PLUS:
            return self.visit(node.left) + self.visit(node.right)
        elif node.op.type == MINUS:
            return self.visit(node.left) - self.visit(node.right)
        elif node.op.type == MUL:
            return self.visit(node.left) * self.visit(node.right)
        elif node.op.type == DIV:
            return self.visit(node.left) / self.visit(node.right)

    def visit_Num(self, node):
        return node.value
```

There are two interesting things about the code that are worth mentioning here: First, the visitor code that manipulates AST nodes is decoupled from the AST nodes themselves. You can see that none of the AST node classes (BinOp and Num) provide any code to manipulate the data stored in those nodes. That logic is encapsulated in the *Interpreter* class that implements the *NodeVisitor* class.

Second, instead of a giant *if* statement in the NodeVisitor's *visit* method like this:

```
def visit(node):
    node_type = type(node).__name__
    if node_type == 'BinOp':
        return self.visit_BinOp(node)
    elif node_type == 'Num':
        return self.visit_Num(node)
    elif ...
    # ...
```

or like this:

```
def visit(node):
    if isinstance(node, BinOp):
        return self.visit_BinOp(node)
    elif isinstance(node, Num):
        return self.visit_Num(node)
    elif ...
```

the NodeVisitor's *visit* method is very generic and dispatches calls to the appropriate method based on the node type passed to it. As I've mentioned before, in order to make use of it, our interpreter inherits from the *NodeVisitor* class and implements necessary methods. So if the type of a node passed to the *visit* method is BinOp, then the *visit* method will dispatch the call to the *visit_BinOp* method, and if the type of a node is Num, then the *visit* method will dispatch the call to the *visit_Num* method, and so on.

Spend some time studying this approach (standard Python module ast (https://docs.python.org/2.7/library/ast.html#module-ast) uses the same mechanism for node traversal) as we will be extending our interpreter with many new *visit_NodeType* methods in the future.

The *generic_visit* method is a fallback that raises an exception to indicate that it encountered a node that the implementation class has no corresponding *visit_NodeType* method for.

Now, let's manually build an AST for the expression 2 * 7 + 3 and pass it to our interpreter to see the visit method in action to evaluate the expression. Here is how you can do it from the Python shell:

```
>>> from spi import Token, MUL, PLUS, INTEGER, Num, BinOp
>>>
>>> mul_token = Token(MUL, '*')
>>> plus_token = Token(PLUS, '+')
>>> mul_node = BinOp(
...     left=Num(Token(INTEGER, 2)),
...     op=mul_token,
...     right=Num(Token(INTEGER, 7))
... )
>>> add_node = BinOp(
...     left=mul_node,
...     op=plus_token,
...     right=Num(Token(INTEGER, 3))
... )
>>> from spi import Interpreter
>>> inter = Interpreter(None)
>>> inter.visit(add_node)
17
```

As you can see, I passed the root of the expression tree to the *visit* method and that triggered traversal of the tree by dispatching calls to the correct methods of the *Interpreter* class(*visit_BinOp* and *visit_Num*) and generating the result.

Okay, here is the complete code of our new interpreter for your convenience:

```
""" SPI - Simple Pascal Interpreter """

###############################################################################
#                                                                             #
#  LEXER                                                                      #
#                                                                             #
###############################################################################

# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, MINUS, MUL, DIV, LPAREN, RPAREN, EOF = (
    'INTEGER', 'PLUS', 'MINUS', 'MUL', 'DIV', '(', ')', 'EOF'
)


class Token(object):
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS, '+')
            Token(MUL, '*')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


class Lexer(object):
    def __init__(self, text):
        # client string input, e.g. "4 + 2 * 3 - 6 / 2"
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Invalid character')

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]
```

```
    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == '+':
                self.advance()
                return Token(PLUS, '+')

            if self.current_char == '-':
                self.advance()
                return Token(MINUS, '-')

            if self.current_char == '*':
                self.advance()
                return Token(MUL, '*')

            if self.current_char == '/':
                self.advance()
                return Token(DIV, '/')

            if self.current_char == '(':
                self.advance()
                return Token(LPAREN, '(')

            if self.current_char == ')':
                self.advance()
                return Token(RPAREN, ')')

            self.error()

        return Token(EOF, None)
```

```
###############################################################################
#                                                                             #
#  PARSER                                                                     #
#                                                                             #
###############################################################################

class AST(object):
    pass


class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right


class Num(AST):
    def __init__(self, token):
        self.token = token
        self.value = token.value


class Parser(object):
    def __init__(self, lexer):
        self.lexer = lexer
        # set current token to the first token taken from the input
        self.current_token = self.lexer.get_next_token()

    def error(self):
        raise Exception('Invalid syntax')

    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            self.current_token = self.lexer.get_next_token()
        else:
            self.error()

    def factor(self):
        """factor : INTEGER | LPAREN expr RPAREN"""
        token = self.current_token
        if token.type == INTEGER:
            self.eat(INTEGER)
            return Num(token)
        elif token.type == LPAREN:
            self.eat(LPAREN)
            node = self.expr()
            self.eat(RPAREN)
            return node

    def term(self):
        """term : factor ((MUL | DIV) factor)*"""
        node = self.factor()
```

```
        while self.current_token.type in (MUL, DIV):
            token = self.current_token
            if token.type == MUL:
                self.eat(MUL)
            elif token.type == DIV:
                self.eat(DIV)

            node = BinOp(left=node, op=token, right=self.factor())

        return node

    def expr(self):
        """
        expr   : term ((PLUS | MINUS) term)*
        term   : factor ((MUL | DIV) factor)*
        factor : INTEGER | LPAREN expr RPAREN
        """
        node = self.term()

        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
            elif token.type == MINUS:
                self.eat(MINUS)

            node = BinOp(left=node, op=token, right=self.term())

        return node

    def parse(self):
        return self.expr()


###############################################################################
#                                                                             #
#  INTERPRETER                                                                #
#                                                                             #
###############################################################################

class NodeVisitor(object):
    def visit(self, node):
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception('No visit_{} method'.format(type(node).__name__))


class Interpreter(NodeVisitor):
    def __init__(self, parser):
        self.parser = parser

    def visit_BinOp(self, node):
        if node.op.type == PLUS:
```

```
            return self.visit(node.left) + self.visit(node.right)
        elif node.op.type == MINUS:
            return self.visit(node.left) - self.visit(node.right)
        elif node.op.type == MUL:
            return self.visit(node.left) * self.visit(node.right)
        elif node.op.type == DIV:
            return self.visit(node.left) / self.visit(node.right)

    def visit_Num(self, node):
        return node.value

    def interpret(self):
        tree = self.parser.parse()
        return self.visit(tree)


def main():
    while True:
        try:
            try:
                text = raw_input('spi> ')
            except NameError:  # Python3
                text = input('spi> ')
        except EOFError:
            break
        if not text:
            continue

        lexer = Lexer(text)
        parser = Parser(lexer)
        interpreter = Interpreter(parser)
        result = interpreter.interpret()
        print(result)


if __name__ == '__main__':
    main()
```

Save the above code into the *spi.py* file or download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part7/python/spi.py). Try it out and see for yourself that your new tree-based interpreter properly evaluates arithmetic expressions.

Here is a sample session:

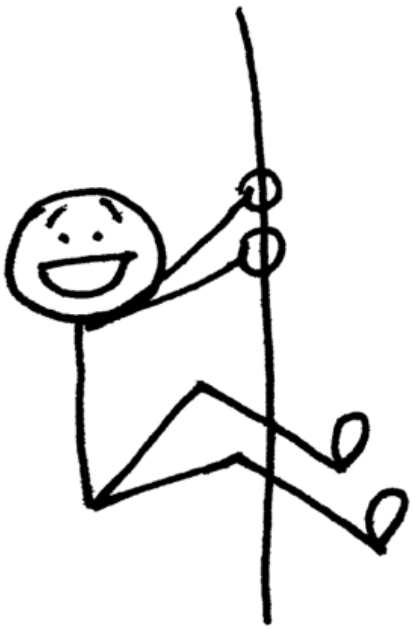
```
$ python spi.py
spi> 7 + 3 * (10 / (12 / (3 + 1) - 1))
22
spi> 7 + 3 * (10 / (12 / (3 + 1) - 1)) / (2 + 3) - 5 - 3 + (8)
10
spi> 7 + (((3 + 2)))
12
```

Today you've learned about parse trees, ASTs, how to construct ASTs and how to traverse them to interpret the input represented by those ASTs. You've also modified the parser and the interpreter and split them apart. The current interface between the lexer, parser, and the interpreter now looks like this:

You can read that as "The parser gets tokens from the lexer and then returns the generated AST for the interpreter to traverse and interpret the input."

That's it for today, but before wrapping up I'd like to talk briefly about recursive-descent parsers, namely just give them a definition because I promised last time to talk about them in more detail. So here you go: a *recursive-descent parser* is a top-down parser that uses a set of recursive procedures to process the input. Top-down reflects the fact that the parser begins by constructing the top node of the parse tree and then gradually constructs lower nodes.

And now it's time for exercises :)



- Write a translator (hint: node visitor) that takes as input an arithmetic expression and prints it out in postfix notation, also known as Reverse Polish Notation (RPN). For example, if the input to the translator is the expression (5 + 3) * 12 / 3 than the output should be 5 3 + 12 * 3 /. See the answer here (https://github.com/rspivak/lsbasi/blob/master/part7/python/ex1.py) but try to solve it first on your own.
- Write a translator (node visitor) that takes as input an arithmetic expression and prints it out in LISP style notation, that is 2 + 3 would become (+ 2 3) and (2 + 3 * 5) would become (+ 2 (* 3 5)). You

can find the answer here (https://github.com/rspivak/lsbasi/blob/master/part7/python/ex2.py) but again try to solve it first before looking at the provided solution.

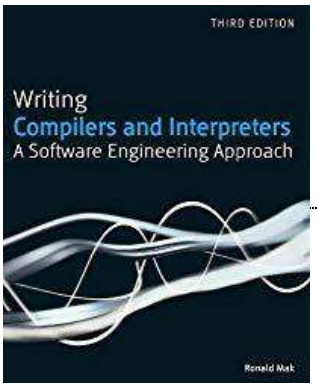
In the next article, we'll add assignment and unary operators to our growing Pascal interpreter. Until then, have fun and see you soon.

P.S. I've also provided a Rust implementation of the interpreter that you can find on GitHub (https://github.com/rspivak/lsbasi/blob/master/part7/rust/spi/src/main.rs). This is a way for me to learn Rust (https://www.rust-lang.org/) so keep in mind that the code might not be "idiomatic" yet. Comments and suggestions as to how to make the code better are always welcome.

Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

   (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=FYAZBCVOB66PGR6J)
3. Modern Compiler Implementation in Java (http://www.amazon.com/gp/product/052182060X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=052182060X&linkCode=as2&tag=russblo0b-20&linkId=ZSKKZMV7YWR22NMW)

Let's Build A Simple Interpreter. Part 8. (https://ruslanspivak.com/lsbasi-part8/)

Date Mon, January 18, 2016

Today we'll talk about **unary operators**, namely unary plus (+) and unary minus (-) operators.

A lot of today's material is based on the material from the previous article, so if you need a refresher just head back to Part 7 (http://ruslanspivak.com/lsbasi-part7/) and go over it again. Remember: repetition is the mother of all learning.

Having said that, this is what you are going to do today:

- extend the grammar to handle unary plus and unary minus operators
- add a new *UnaryOp* AST node class
- extend the parser to generate an AST with *UnaryOp* nodes
- extend the interpreter and add a new *visit_UnaryOp* method to interpret unary operators

Let's get started, shall we?

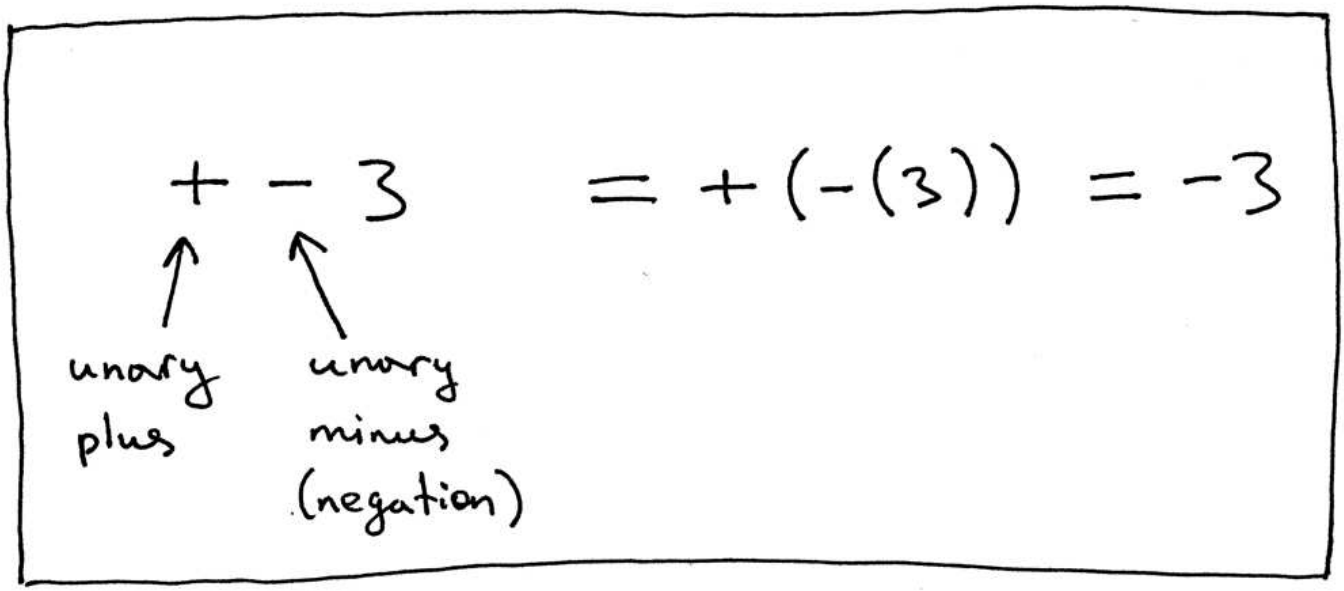
So far we've worked with binary operators only (+, -, *, /), that is, the operators that operate on two operands.

What is a unary operator then? A *unary operator* is an operator that operates on one *operand* only.

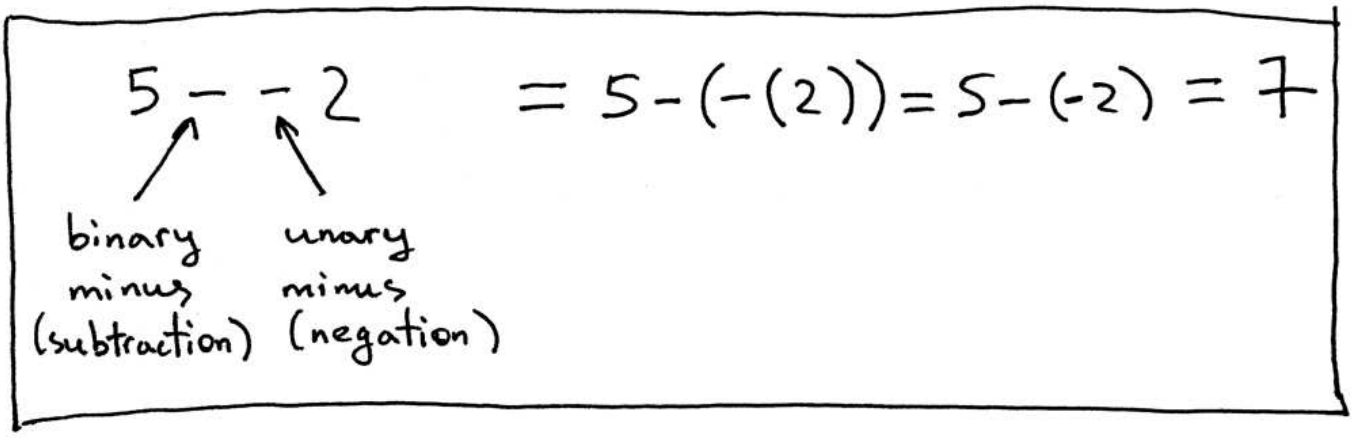
Here are the rules for unary plus and unary minus operators:

- The unary minus (-) operator produces the negation of its numeric operand
- The unary plus (+) operator yields its numeric operand without change
- The unary operators have higher precedence than the binary operators +, -, *, and /

In the expression "+ - 3" the first '+' operator represents the unary plus operation and the second '-' operator represents the unary minus operation. The expression "+ - 3" is equivalent to "+ (- (3))" which is equal to -3. One could also say that **-3** in the expression is a negative integer, but in our case we treat it as a unary minus operator with 3 as its positive integer operand:

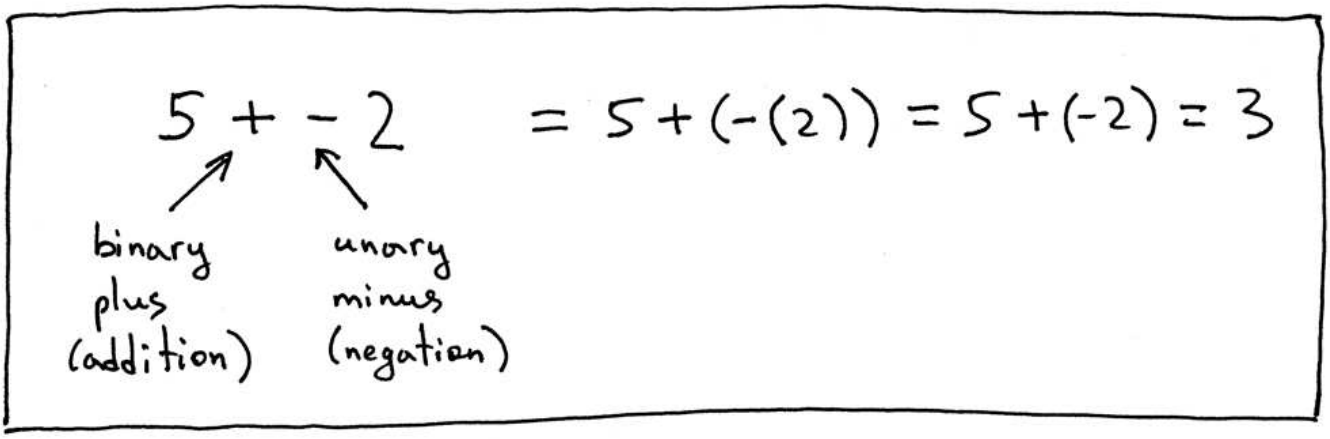


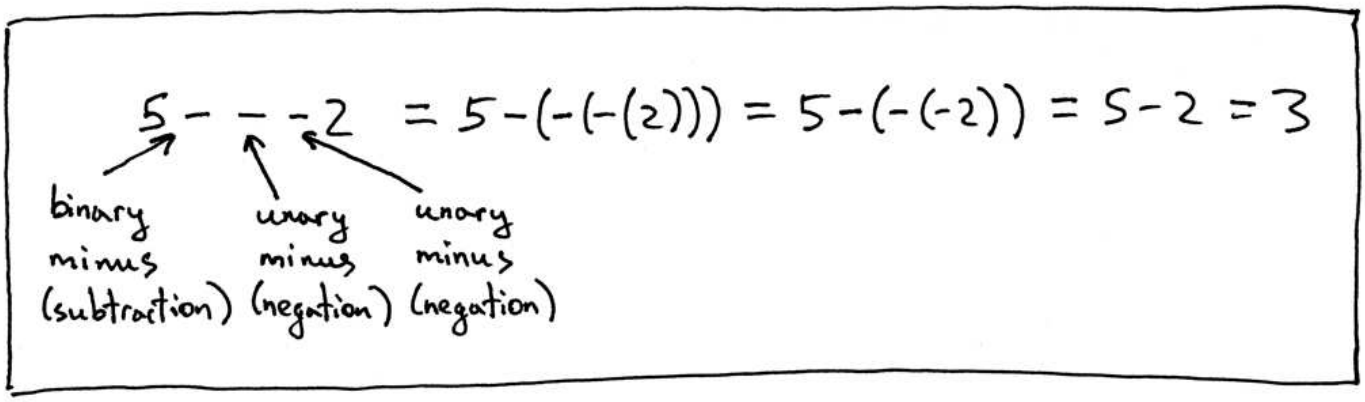
Let's take a look at another expression, "5 - - 2":



In the expression "5 - - 2" the first '-' represents the *binary* subtraction operation and the second '-' represents the *unary* minus operation, the negation.

And some more examples:





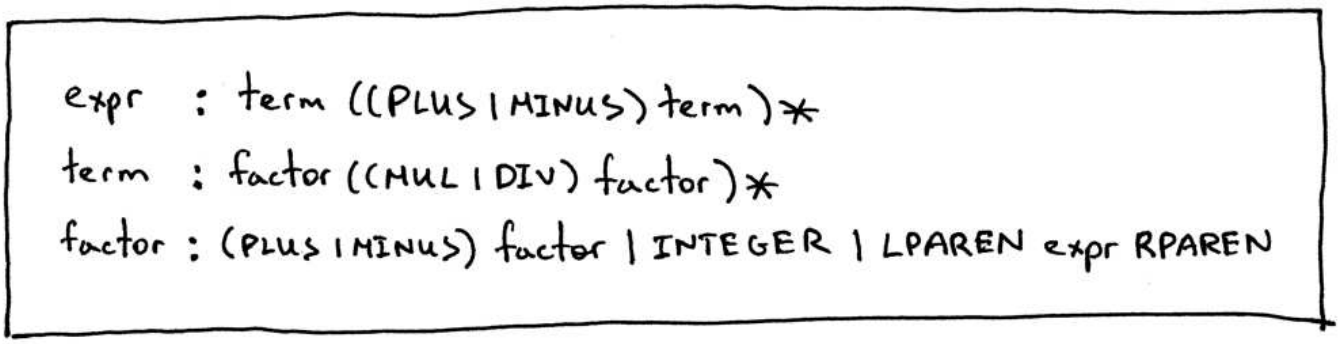
Now let's update our grammar to include unary plus and unary minus operators. We'll modify the *factor* rule and add unary operators there because unary operators have higher precedence than binary +, -, * and / operators.

This is our current *factor* rule:

And this is our updated *factor* rule to handle unary plus and unary minus operators:

As you can see, I extended the *factor* rule to reference itself, which allows us to derive expressions like "- - - + - 3", a legitimate expression with a lot of unary operators.

Here is the full grammar that can now derive expressions with unary plus and unary minus operators:



The next step is to add an AST node class to represent unary operators.

This one will do:

```
class UnaryOp(AST):
    def __init__(self, op, expr):
        self.token = self.op = op
        self.expr = expr
```

The constructor takes two parameters: *op*, which represents the unary operator token (plus or minus) and *expr*, which represents an AST node.

Our updated grammar had changes to the *factor* rule, so that's what we're going to modify in our parser - the *factor* method. We will add code to the method to handle the "(PLUS | MINUS) factor" sub-rule:

```
def factor(self):
    """factor : (PLUS | MINUS) factor | INTEGER | LPAREN expr RPAREN"""
    token = self.current_token
    if token.type == PLUS:
        self.eat(PLUS)
        node = UnaryOp(token, self.factor())
        return node
    elif token.type == MINUS:
        self.eat(MINUS)
        node = UnaryOp(token, self.factor())
        return node
    elif token.type == INTEGER:
        self.eat(INTEGER)
        return Num(token)
    elif token.type == LPAREN:
        self.eat(LPAREN)
        node = self.expr()
        self.eat(RPAREN)
        return node
```

And now we need to extend the *Interpreter* class and add a *visit_UnaryOp* method to interpret unary nodes:

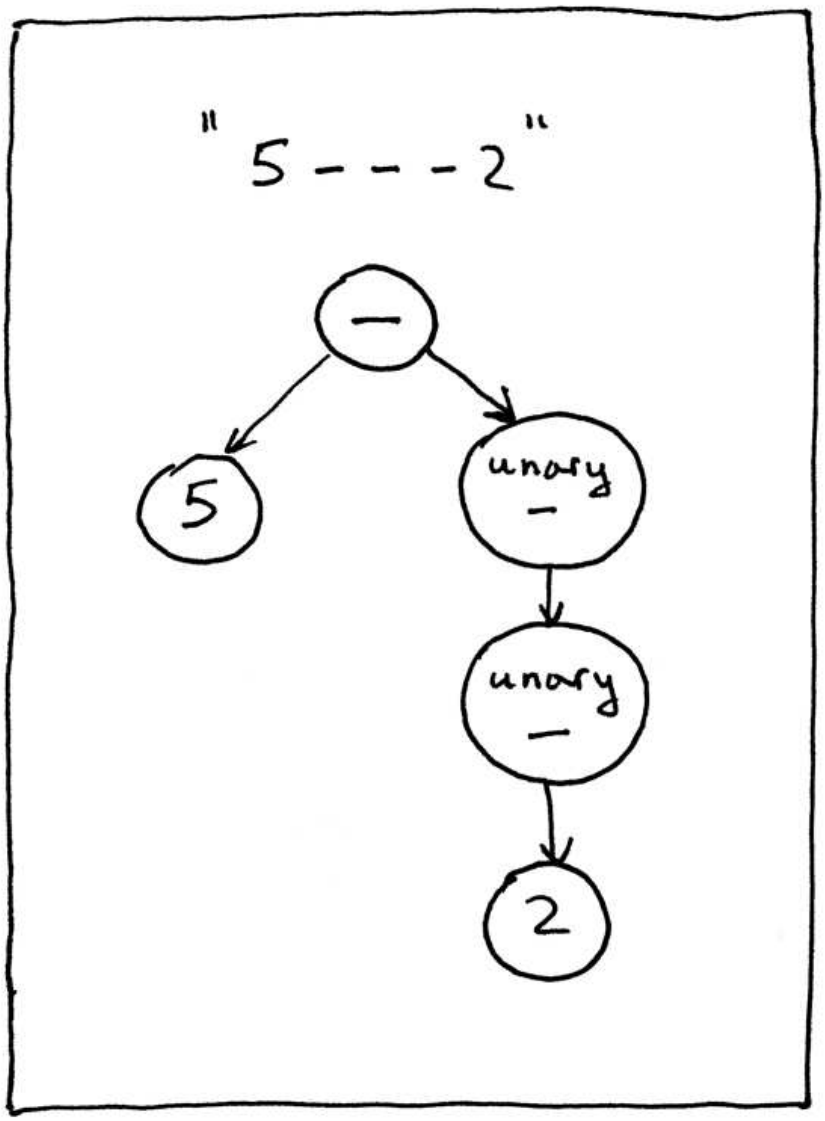
```
def visit_UnaryOp(self, node):
    op = node.op.type
    if op == PLUS:
        return +self.visit(node.expr)
    elif op == MINUS:
        return -self.visit(node.expr)
```

Onward!

Let's manually build an AST for the expression "5 - - - 2" and pass it to our interpreter to verify that the new *visit_UnaryOp* method works. Here is how you can do it from the Python shell:

```
>>> from spi import BinOp, UnaryOp, Num, MINUS, INTEGER, Token
>>> five_tok = Token(INTEGER, 5)
>>> two_tok = Token(INTEGER, 2)
>>> minus_tok = Token(MINUS, '-')
>>> expr_node = BinOp(
...     Num(five_tok),
...     minus_tok,
...     UnaryOp(minus_token, UnaryOp(minus_token, Num(two_tok)))
... )
>>> from spi import Interpreter
>>> inter = Interpreter(None)
>>> inter.visit(expr_node)
3
```

Visually the above AST tree looks like this:



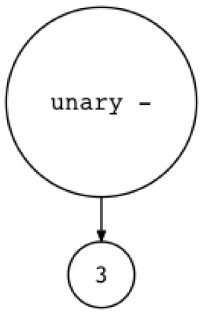
Download the full source code of the interpreter for this article directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part8/python/spi.py). Try it out and see for yourself that your updated tree-based interpreter properly evaluates arithmetic expressions containing unary operators.

Here is a sample session:

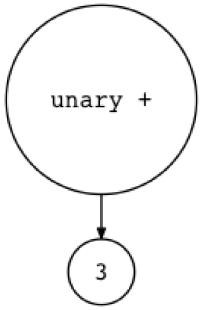
```
$ python spi.py
spi> - 3
-3
spi> + 3
3
spi> 5 - - - + - 3
8
spi> 5 - - - + - (3 + 4) - +2
10
```

I also updated the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part8/python/genastdot.py) utility to handle unary operators. Here are some of the examples of the generated AST images for expressions with unary operators:

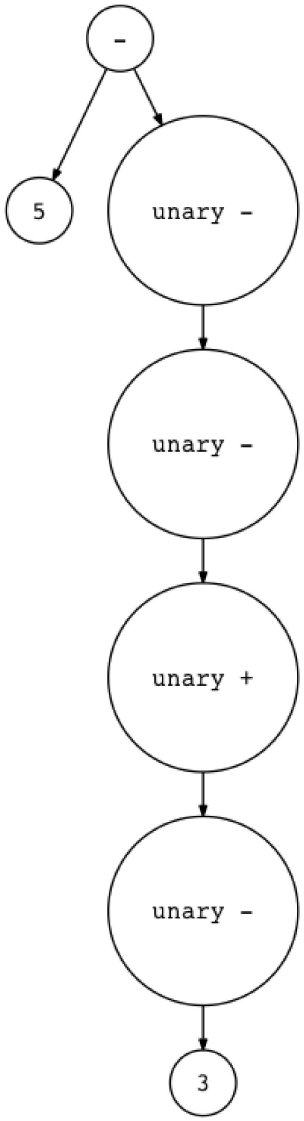
```
$ python genastdot.py "- 3" > ast.dot && dot -Tpng -o ast.png ast.dot
```



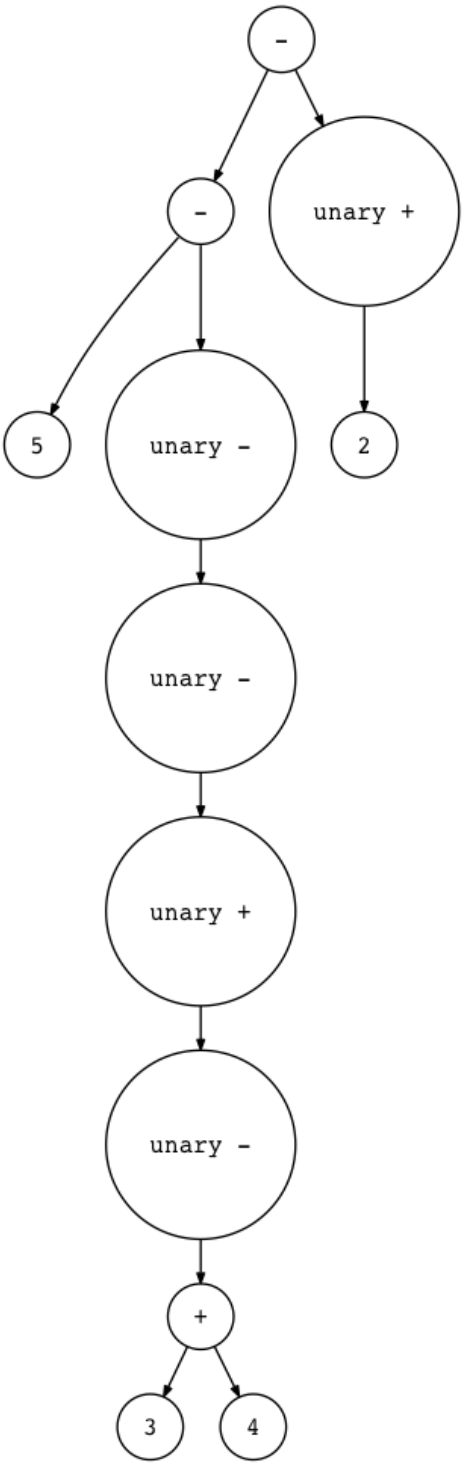

```
$ python genastdot.py "+ 3" > ast.dot && dot -Tpng -o ast.png ast.dot
```



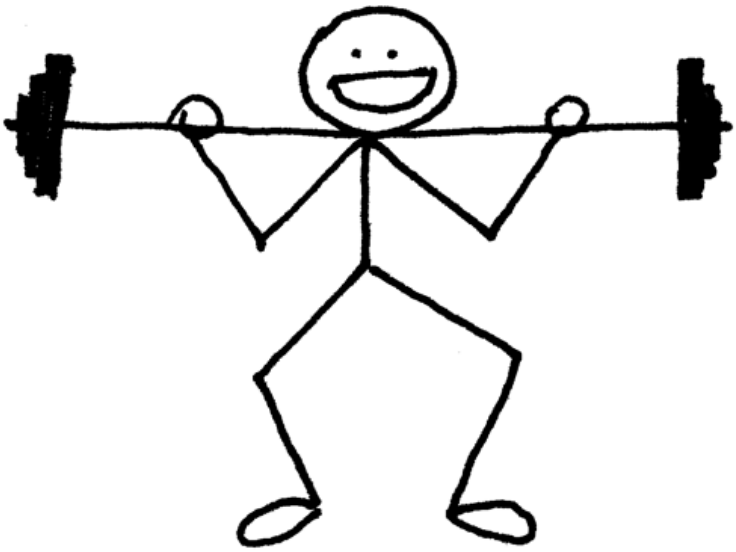

```
$ python genastdot.py "5 - - - + - 3" > ast.dot && dot -Tpng -o ast.png ast.dot
```




```
$ python genastdot.py "5 - - - + - (3 + 4) - +2" \
  > ast.dot && dot -Tpng -o ast.png ast.dot
```



And here is a new exercise for you:



- Install Free Pascal (http://www.freepascal.org/), compile and run testunary.pas (https://github.com/rspivak/lsbasi/blob/master/part8/python/testunary.pas), and verify that the results are the same as produced with your spi (https://github.com/rspivak/lsbasi/blob/master/part8/python/spi.py) interpreter.

That's all for today. In the next article, we'll tackle assignment statements. Stay tuned and see you soon.

Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

   (http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
2. Writing Compilers and Interpreters: A Software Engineering Approach (http://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=UCLGQTPIYSWYKRRM)

Let's Build A Simple Interpreter. Part 9. (https://ruslanspivak.com/lsbasi-part9/)

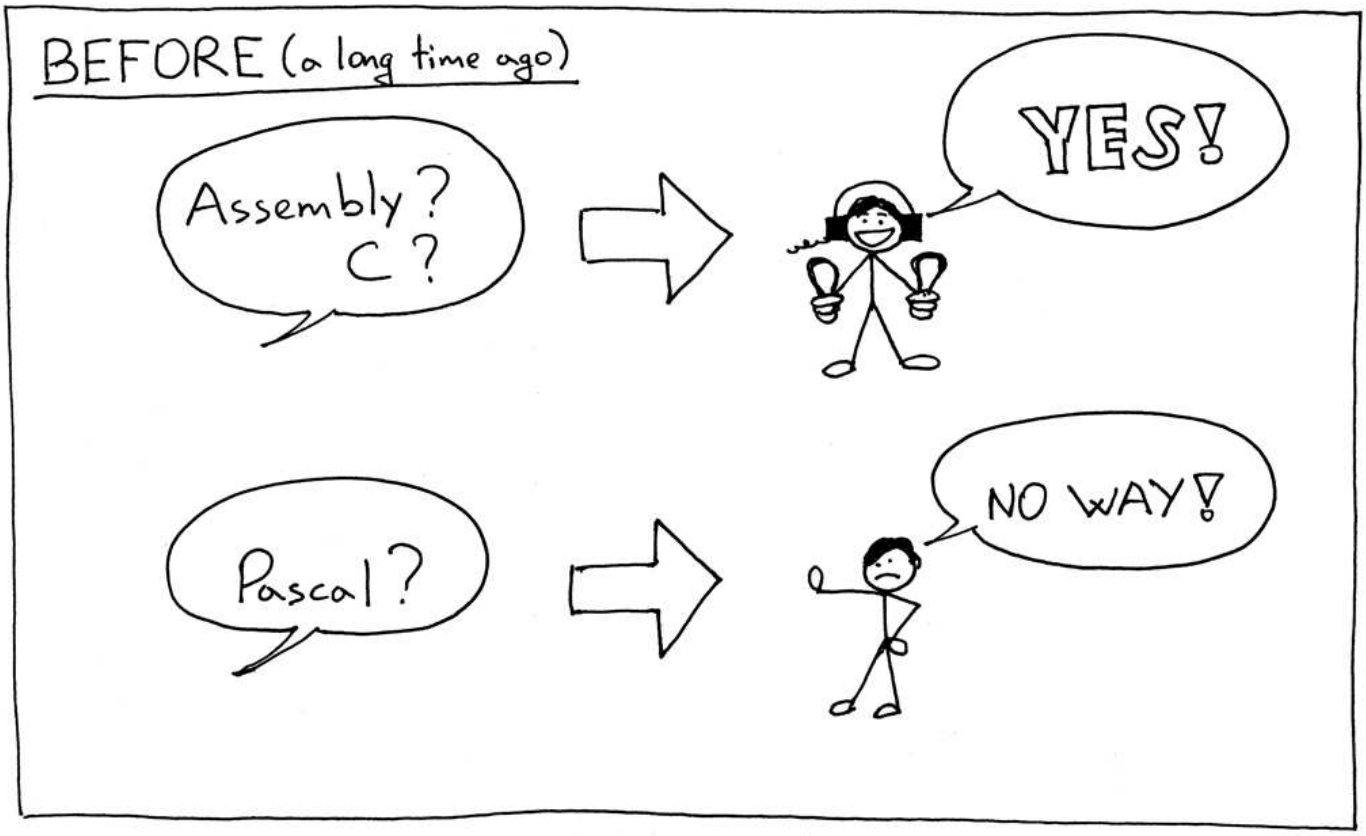
Date Sun, May 01, 2016

I remember when I was in university (a long time ago) and learning systems programming, I believed that the only "real" languages were Assembly and C. And Pascal was - how to put it nicely - a very high-level language used by application developers who didn't want to know what was going on under the hood.

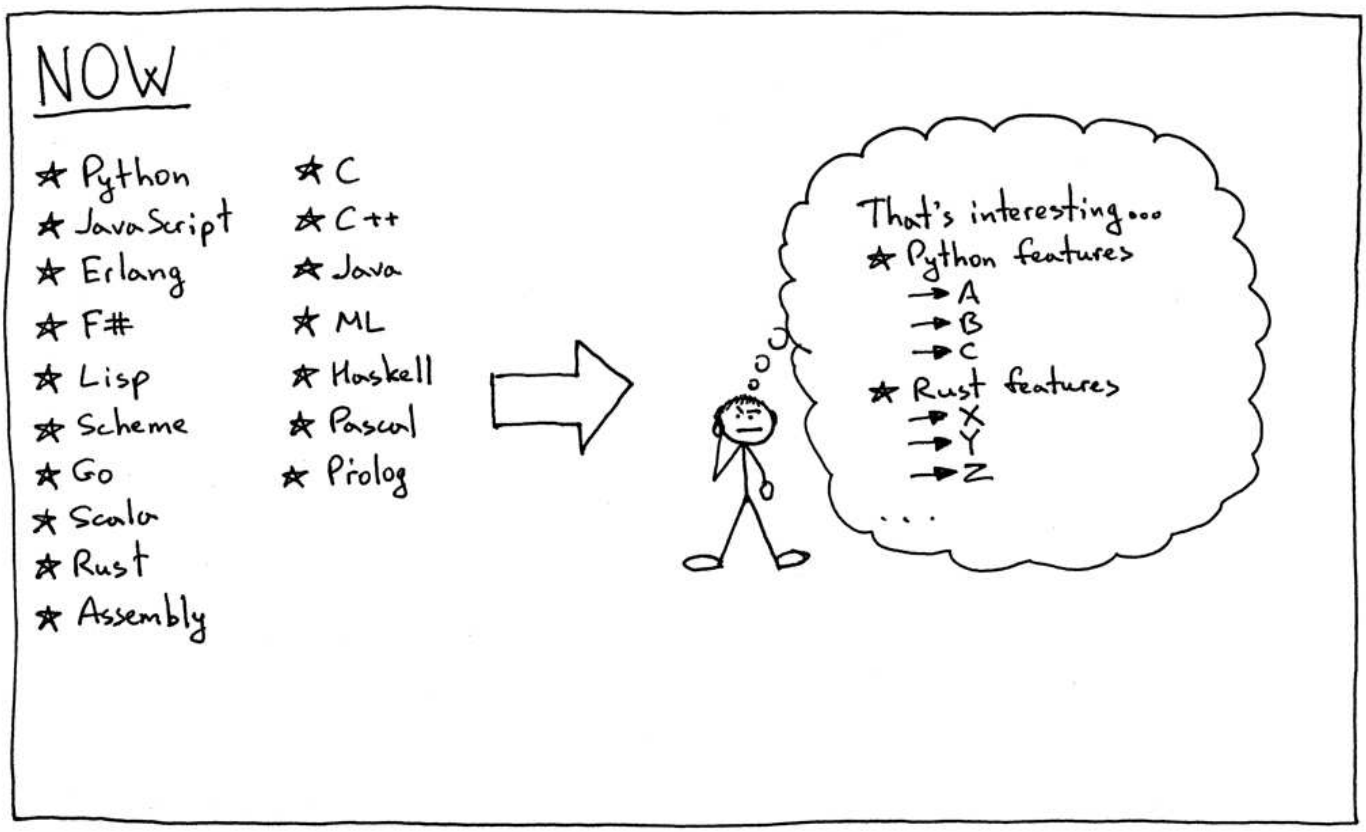
Little did I know back then that I would be writing almost everything in Python (and love every bit of it) to pay my bills and that I would also be writing an interpreter and compiler for Pascal for the reasons I stated in the very first article of the series (/lsbasi-part1/).

These days, I consider myself a programming languages enthusiast, and I'm fascinated by all languages and their unique features. Having said that, I have to note that I enjoy using certain languages way more than others. I am biased and I'll be the first one to admit that. :)

This is me before:



And now:



Okay, let's get down to business. Here is what you're going to learn today:

1. How to parse and interpret a Pascal program definition.
2. How to parse and interpret compound statements.
3. How to parse and interpret assignment statements, including variables.
4. A bit about symbol tables and how to store and lookup variables.

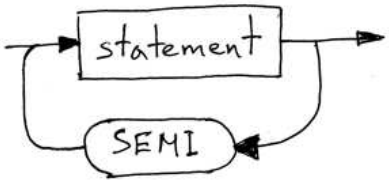
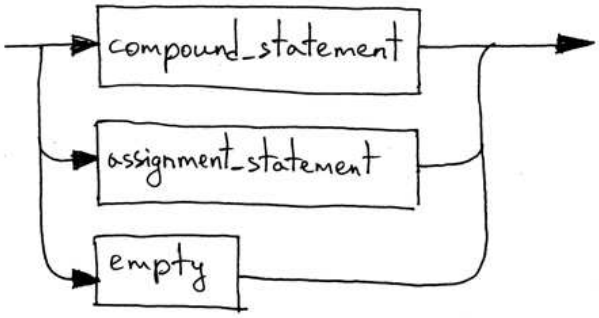
I'll use the following sample Pascal-like program to introduce new concepts:

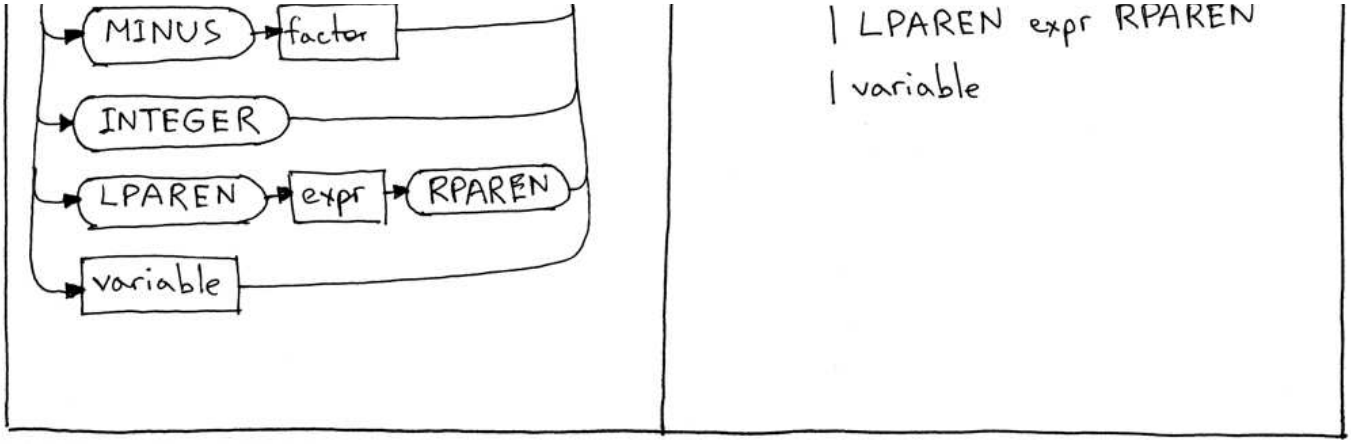
```
BEGIN
    BEGIN
        number := 2;
        a := number;
        b := 10 * a + 10 * number / 4;
        c := a - - b
    END;
    x := 11;
END.
```

You could say that that's quite a jump from the command line interpreter you wrote so far by following the previous articles in the series, but it's a jump that I hope will bring excitement. It's not "just" a calculator anymore, we're getting serious here, Pascal serious. :)

Let's dive in and look at syntax diagrams for new language constructs and their corresponding grammar rules.

On your marks: Ready. Set. Go!

SYNTAX DIAGRAM	GRAMMAR RULE
1 program 	program : compound_statement DOT
2 compound_statement 	compound_statement: BEGIN statement_list END
3 statement_list 	statement_list: statement | statement SEMI statement_list
4 statement 	statement : compound_statement | assignment_statement | empty
5 assignment_statement 	assignment_statement: variable ASSIGN expr
6 variable 	variable: ID
7 empty 	empty :
8 factor 	factor : PLUS factor | MINUS factor | INTEGER



1. I'll start with describing what a Pascal *program* is. A Pascal ***program*** consists of a *compound statement* that ends with a dot. Here is an example of a program:

   ```
   "BEGIN  END."
   ```

   I have to note that this is not a complete program definition, and we'll extend it later in the series.

2. What is a *compound statement*? A ***compound statement*** is a block marked with BEGIN and END that can contain a list (possibly empty) of statements including other compound statements. Every statement inside the compound statement, except for the last one, must terminate with a semicolon. The last statement in the block may or may not have a terminating semicolon. Here are some examples of valid compound statements:

   ```
   "BEGIN END"
   "BEGIN a := 5; x := 11 END"
   "BEGIN a := 5; x := 11; END"
   "BEGIN BEGIN a := 5 END; x := 11 END"
   ```

3. A ***statement list*** is a list of zero or more statements inside a compound statement. See above for some examples.

4. A ***statement*** can be a *compound statement*, an *assignment statement*, or it can be an *empty* statement.

5. An ***assignment statement*** is a variable followed by an ASSIGN token (two characters, ':' and '=') followed by an expression.

   ```
   "a := 11"
   "b := a + 9 - 5 * 2"
   ```

6. A ***variable*** is an identifier. We'll use the ID token for variables. The value of the token will be a variable's name like 'a', 'number', and so on. In the following code block 'a' and 'b' are variables:

   ```
   "BEGIN a := 11; b := a + 9 - 5 * 2 END"
   ```

7. An ***empty*** statement represents a grammar rule with no further productions. We use the *empty_statement* grammar rule to indicate the end of the *statement_list* in the parser and also to allow for empty compound statements as in 'BEGIN END'.

8. The ***factor*** rule is updated to handle variables.

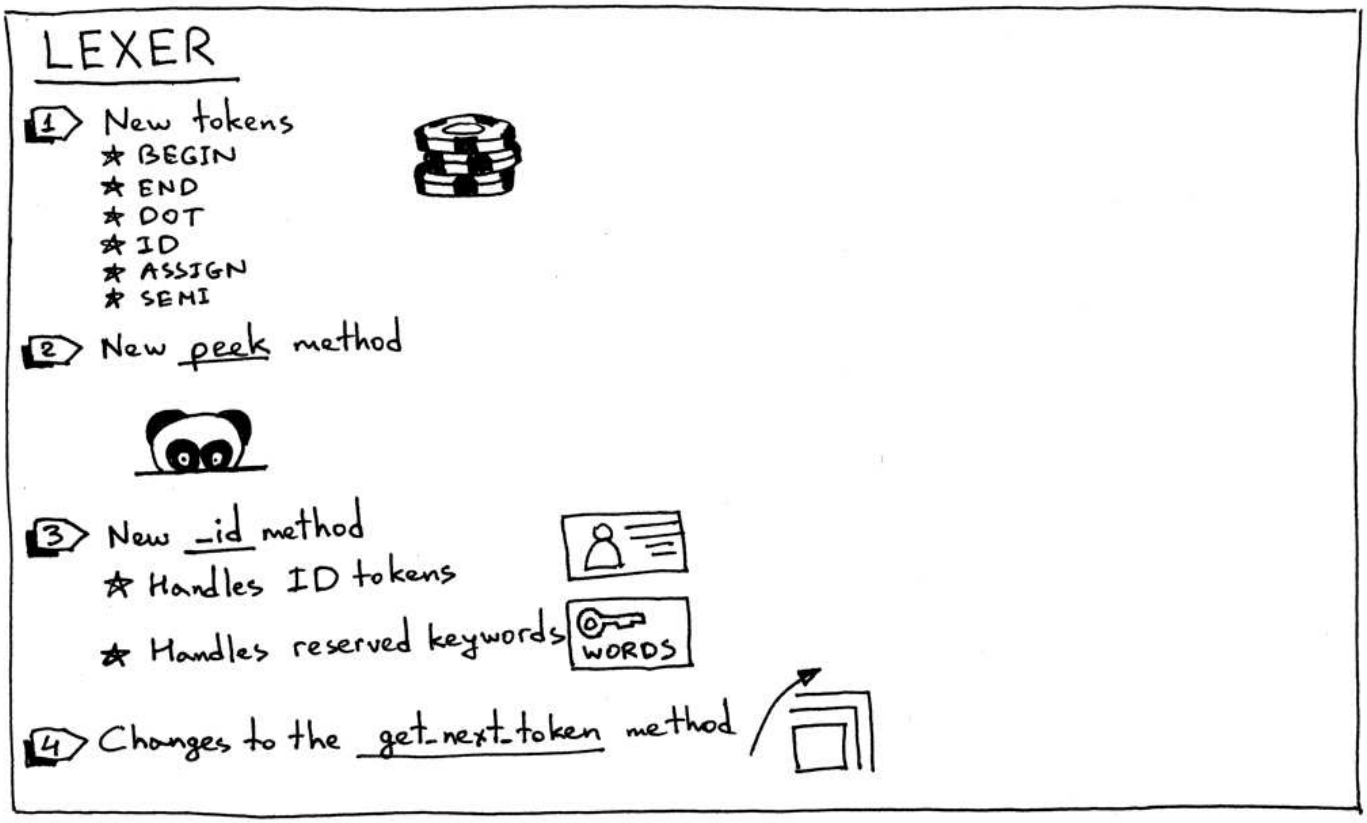
Now let's take a look at our complete grammar:

```
program : compound_statement DOT

compound_statement : BEGIN statement_list END

statement_list : statement
               | statement SEMI statement_list

statement : compound_statement
          | assignment_statement
          | empty

assignment_statement : variable ASSIGN expr

empty :

expr: term ((PLUS | MINUS) term)*

term: factor ((MUL | DIV) factor)*

factor : PLUS factor
       | MINUS factor
       | INTEGER
       | LPAREN expr RPAREN
       | variable

variable: ID
```

You probably noticed that I didn't use the star **'*'** symbol in the *compound_statement* rule to represent zero or more repetitions, but instead explicitly specified the *statement_list* rule. This is another way to represent the 'zero or more' operation, and it will come in handy when we look at parser generators like PLY (http://www.dabeaz.com/ply/), later in the series. I also split the "(PLUS | MINUS) factor" sub-rule into two separate rules.

In order to support the updated grammar, we need to make a number of changes to our lexer, parser, and interpreter. Let's go over those changes one by one.

Here is the summary of the changes in our lexer:



1. To support a Pascal program's definition, compound statements, assignment statements, and variables, our lexer needs to return new tokens:
    - BEGIN (to mark the beginning of a compound statement)
    - END (to mark the end of the compound statement)
    - DOT (a token for a dot character '.' required by a Pascal program's definition)
    - ASSIGN (a token for a two character sequence ':='). In Pascal, an assignment operator is different than in many other languages like C, Python, Java, Rust, or Go, where you would use single character '=' to indicate assignment
    - SEMI (a token for a semicolon character ';' that is used to mark the end of a statement inside a compound statement)
    - ID (A token for a valid identifier. Identifiers start with an alphabetical character followed by any number of alphanumerical characters)
2. Sometimes, in order to be able to differentiate between different tokens that start with the same character, (':' vs ':=' or '==' vs '=>') we need to peek into the input buffer without actually consuming the next character. For this particular purpose, I introduced a *peek* method that will help us tokenize assignment statements. The method is not strictly required, but I thought I would introduce it earlier in the series and it will also make the *get_next_token* method a bit cleaner. All it does is return the next character from the text buffer without incrementing the *self.pos* variable. Here is the method itself:

```
def peek(self):
    peek_pos = self.pos + 1
    if peek_pos > len(self.text) - 1:
        return None
    else:
        return self.text[peek_pos]
```

3. Because Pascal variables and reserved keywords are both identifiers, we will combine their handling into one method called *_id*. The way it works is that the lexer consumes a sequence of alphanumerical characters and then checks if the character sequence is a reserved word. If it is, it returns a pre-constructed token for that reserved keyword. And if it's not a reserved keyword, it returns a new ID token whose value is the character string (lexeme). I bet at this point you think, "Gosh, just show me the code." :) Here it is:

```
RESERVED_KEYWORDS = {
    'BEGIN': Token('BEGIN', 'BEGIN'),
    'END': Token('END', 'END'),
}

def _id(self):
    """Handle identifiers and reserved keywords"""
    result = ''
    while self.current_char is not None and self.current_char.isalnum():
        result += self.current_char
        self.advance()

    token = RESERVED_KEYWORDS.get(result, Token(ID, result))
    return token
```

4. And now let's take a look at the changes in the main lexer method *get_next_token*:

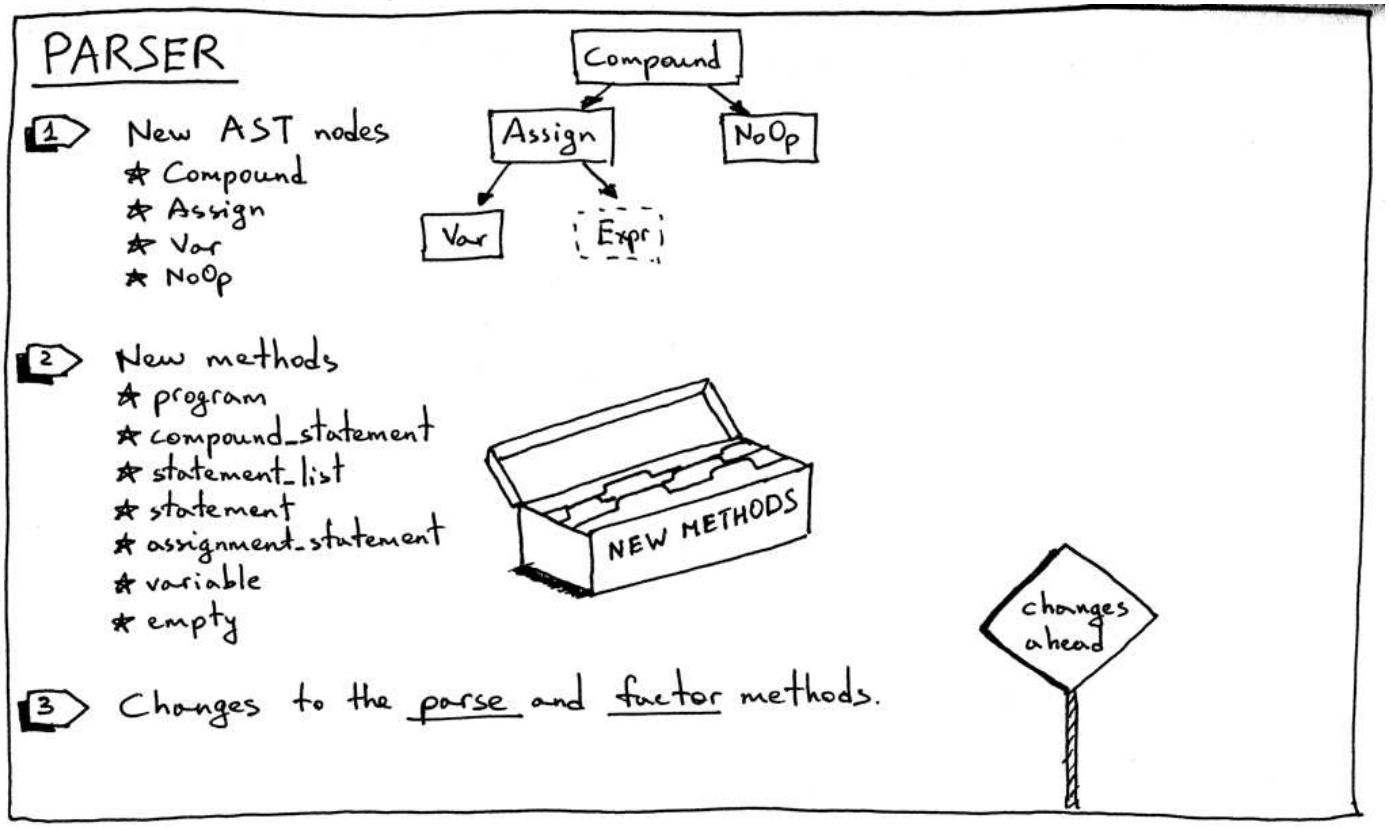
```
def get_next_token(self):
    while self.current_char is not None:
        ...
        if self.current_char.isalpha():
            return self._id()

        if self.current_char == ':' and self.peek() == '=':
            self.advance()
            self.advance()
            return Token(ASSIGN, ':=')

        if self.current_char == ';':
            self.advance()
            return Token(SEMI, ';')

        if self.current_char == '.':
            self.advance()
            return Token(DOT, '.')
        ...
```

It's time to see our shiny new lexer in all its glory and action. Download the source code from GitHub (https://github.com/rspivak/lsbasi/blob/master/part9/python) and launch your Python shell from the same directory where you saved the spi.py (https://github.com/rspivak/lsbasi/blob/master/part9/python/spi.py) file:

```
>>> from spi import Lexer
>>> lexer = Lexer('BEGIN a := 2; END.')
>>> lexer.get_next_token()
Token(BEGIN, 'BEGIN')
>>> lexer.get_next_token()
Token(ID, 'a')
>>> lexer.get_next_token()
Token(ASSIGN, ':=')
>>> lexer.get_next_token()
Token(INTEGER, 2)
>>> lexer.get_next_token()
Token(SEMI, ';')
>>> lexer.get_next_token()
Token(END, 'END')
>>> lexer.get_next_token()
Token(DOT, '.')
>>> lexer.get_next_token()
Token(EOF, None)
>>>
```

Moving on to parser changes.

Here is the summary of changes in our parser:



1. Let's start with new AST nodes:
    - *Compound* AST node represents a compound statement. It contains a list of statement nodes in its *children* variable.

```
class Compound(AST):
    """Represents a 'BEGIN ... END' block"""
    def __init__(self):
        self.children = []
```

- *Assign* AST node represents an assignment statement. Its *left* variable is for storing a *Var* node and its *right* variable is for storing a node returned by the expr parser method:

```
class Assign(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right
```

- *Var* AST node (you guessed it) represents a variable. The *self.value* holds the variable's name.

```
class Var(AST):
    """The Var node is constructed out of ID token."""
    def __init__(self, token):
        self.token = token
        self.value = token.value
```

- *NoOp* node is used to represent an *empty* statement. For example 'BEGIN END' is a valid compound statement that has no statements.

```
class NoOp(AST):
    pass
```

2. As you remember, each rule from the grammar has a corresponding method in our recursive-descent parser. This time we're adding seven new methods. These methods are responsible for parsing new language constructs and constructing new AST nodes. They are pretty straightforward:

```
def program(self):
    """program : compound_statement DOT"""
    node = self.compound_statement()
    self.eat(DOT)
    return node

def compound_statement(self):
    """
    compound_statement: BEGIN statement_list END
    """
    self.eat(BEGIN)
    nodes = self.statement_list()
    self.eat(END)

    root = Compound()
    for node in nodes:
        root.children.append(node)

    return root

def statement_list(self):
    """
    statement_list : statement
                   | statement SEMI statement_list
    """
    node = self.statement()

    results = [node]

    while self.current_token.type == SEMI:
        self.eat(SEMI)
        results.append(self.statement())

    if self.current_token.type == ID:
        self.error()

    return results

def statement(self):
    """
    statement : compound_statement
              | assignment_statement
              | empty
    """
    if self.current_token.type == BEGIN:
        node = self.compound_statement()
    elif self.current_token.type == ID:
        node = self.assignment_statement()
    else:
        node = self.empty()
    return node

def assignment_statement(self):
    """
    assignment_statement : variable ASSIGN expr
    """
    left = self.variable()
```

```
        token = self.current_token
        self.eat(ASSIGN)
        right = self.expr()
        node = Assign(left, token, right)
        return node

    def variable(self):
        """
        variable : ID
        """
        node = Var(self.current_token)
        self.eat(ID)
        return node

    def empty(self):
        """An empty production"""
        return NoOp()
```

3. We also need to update the existing *factor* method to parse variables:

```
def factor(self):
    """factor : PLUS  factor
              | MINUS factor
              | INTEGER
              | LPAREN expr RPAREN
              | variable
    """
    token = self.current_token
    if token.type == PLUS:
        self.eat(PLUS)
        node = UnaryOp(token, self.factor())
        return node
    ...
    else:
        node = self.variable()
        return node
```

4. The parser's *parse* method is updated to start the parsing process by parsing a program definition:

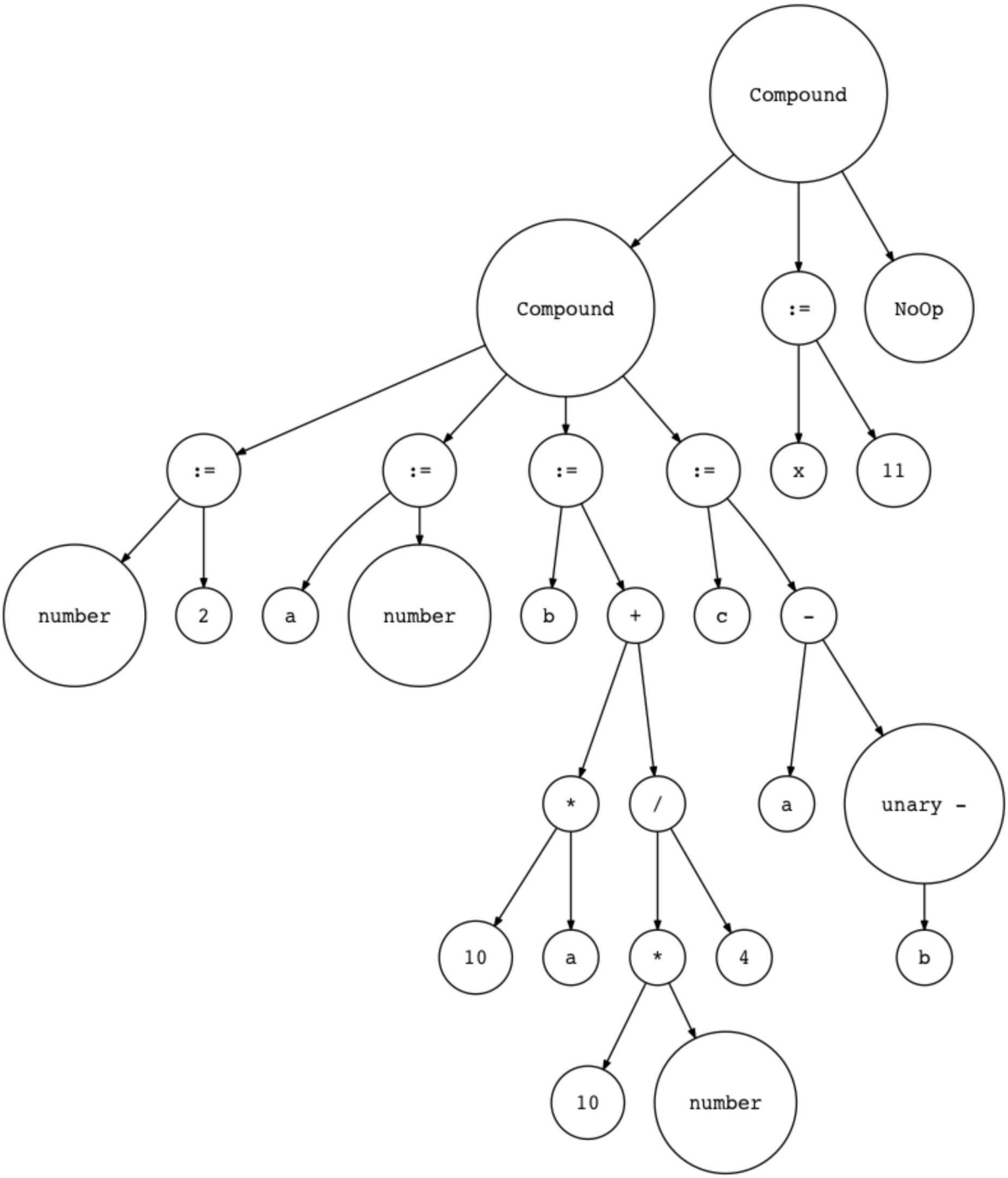
```
def parse(self):
    node = self.program()
    if self.current_token.type != EOF:
        self.error()

    return node
```

Here is our sample program again:

```
BEGIN
    BEGIN
        number := 2;
        a := number;
        b := 10 * a + 10 * number / 4;
        c := a - - b
    END;
    x := 11;
END.
```

Let's visualize it with genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part9/python/genastdot.py) (For brevity, when displaying a *Var* node, it just shows the node's variable name and when displaying an Assign node it shows ':=' instead of showing 'Assign' text):

```
$ python genastdot.py assignments.txt > ast.dot && dot -Tpng -o ast.png ast.dot
```



And finally, here are the required interpreter changes:

To interpret new AST nodes, we need to add corresponding visitor methods to the interpreter. There are four new visitor methods:

- visit_Compound
- visit_Assign
- visit_Var
- visit_NoOp

Compound and *NoOp* visitor methods are pretty straightforward. The *visit_Compound* method iterates over its children and visits each one in turn, and the *visit_NoOp* method does nothing.

```
def visit_Compound(self, node):
    for child in node.children:
        self.visit(child)

def visit_NoOp(self, node):
    pass
```

The *Assign* and *Var* visitor methods deserve a closer examination.

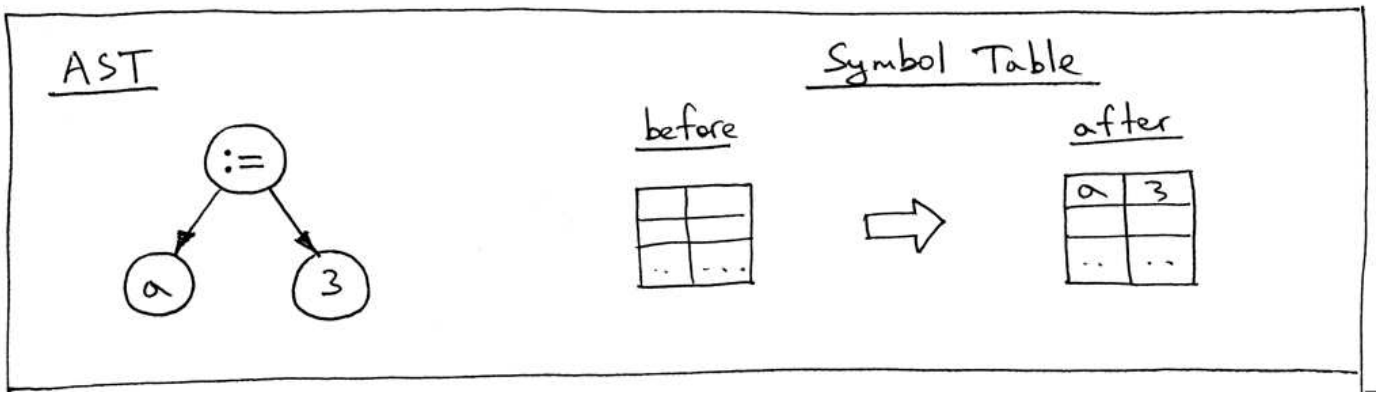
When we assign a value to a variable, we need to store that value somewhere for when we need it later, and that's exactly what the *visit_Assign* method does:

```
def visit_Assign(self, node):
    var_name = node.left.value
    self.GLOBAL_SCOPE[var_name] = self.visit(node.right)
```

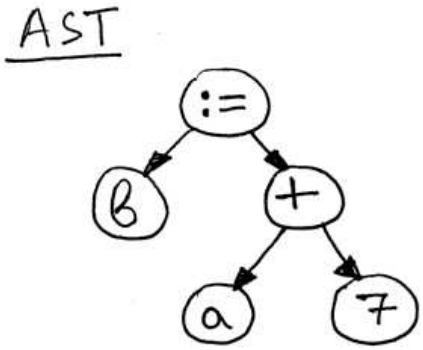
The method stores a key-value pair (a variable name and a value associated with the variable) in a *symbol table* GLOBAL_SCOPE. What is a *symbol table*? A ***symbol table*** is an abstract data type (**ADT**) for tracking various symbols in source code. The only symbol category we have right now is variables

and we use the Python dictionary to implement the symbol table ADT. For now I'll just say that the way the symbol table is used in this article is pretty "hacky": it's not a separate class with special methods but a simple Python dictionary and it also does double duty as a memory space. In future articles, I will be talking about symbol tables in much greater detail, and together we'll also remove all the hacks.

Let's take a look at an AST for the statement "a := 3;" and the symbol table before and after the *visit_Assign* method does its job:



Now let's take a look at an AST for the statement "b := a + 7;"

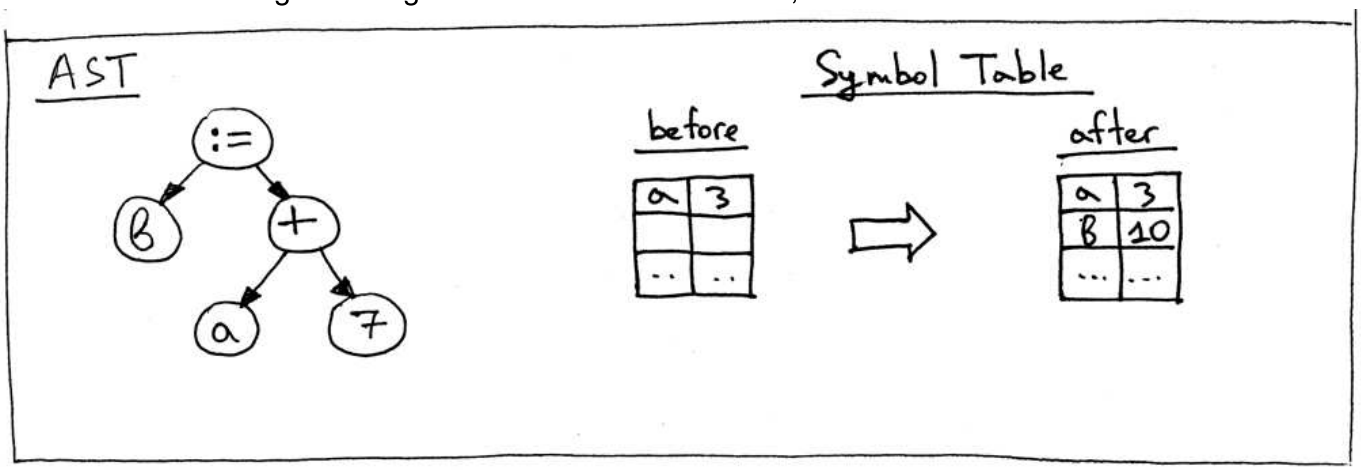


As you can see, the right-hand side of the assignment statement - "a + 7" - references the variable 'a', so before we can evaluate the expression "a + 7" we need to find out what the value of 'a' is and that's the responsibility of the *visit_Var* method:

```
def visit_Var(self, node):
    var_name = node.value
    val = self.GLOBAL_SCOPE.get(var_name)
    if val is None:
        raise NameError(repr(var_name))
    else:
        return val
```

When the method visits a *Var* node as in the above AST picture, it first gets the variable's name and then uses that name as a key into the *GLOBAL_SCOPE* dictionary to get the variable's value. If it can find the value, it returns it, if not - it raises a *NameError* exception. Here are the contents of the symbol

table before evaluating the assignment statement "b := a + 7;":



These are all the changes that we need to do today to make our interpreter tick. At the end of the main program, we simply print the contents of the symbol table GLOBAL_SCOPE to standard output.

Let's take our updated interpreter for a drive both from a Python interactive shell and from the command line. Make sure that you downloaded both the source code for the interpreter and the assignments.txt (https://github.com/rspivak/lsbasi/blob/master/part9/python/assignments.txt) file before testing:

Launch your Python shell:

```
$ python
>>> from spi import Lexer, Parser, Interpreter
>>> text = """\
... BEGIN
...
...     BEGIN
...         number := 2;
...         a := number;
...         b := 10 * a + 10 * number / 4;
...         c := a - - b
...     END;
...
...     x := 11;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> interpreter = Interpreter(parser)
>>> interpreter.interpret()
>>> print(interpreter.GLOBAL_SCOPE)
{'a': 2, 'x': 11, 'c': 27, 'b': 25, 'number': 2}
```

And from the command line, using a source file as input to our interpreter:

```
$ python spi.py assignments.txt
{'a': 2, 'x': 11, 'c': 27, 'b': 25, 'number': 2}
```

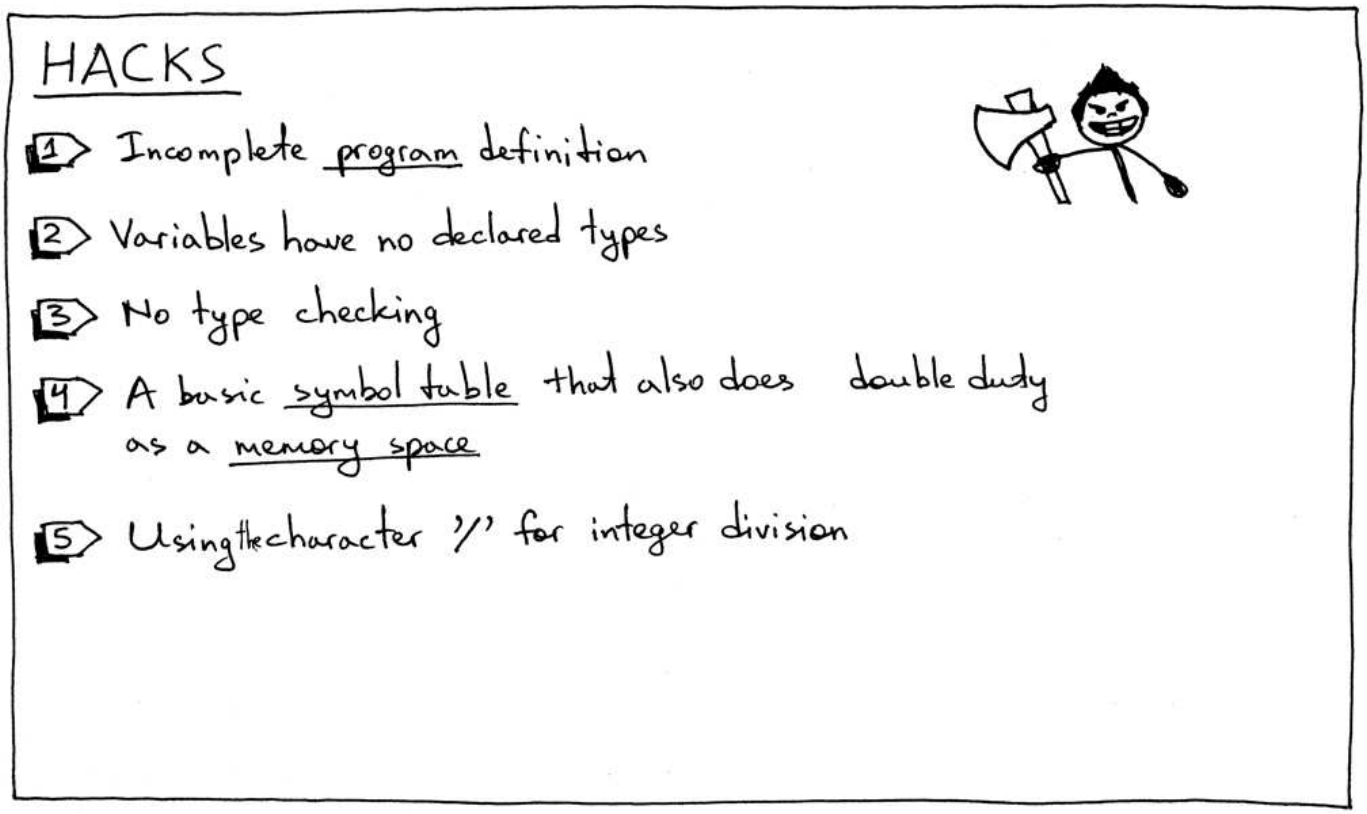
If you haven't tried it yet, try it now and see for yourself that the interpreter is doing its job properly.

Let's sum up what you had to do to extend the Pascal interpreter in this article:

1. Add new rules to the grammar
2. Add new tokens and supporting methods to the lexer and update the *get_next_token* method

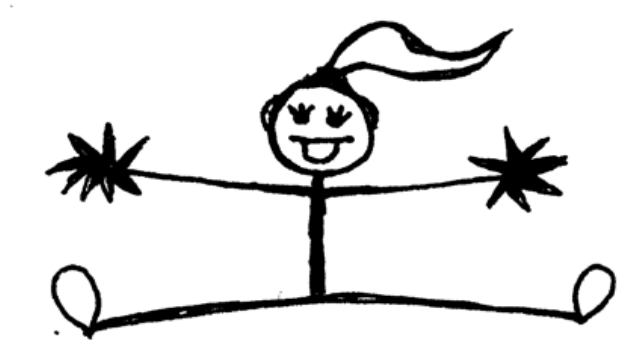
3. Add new AST nodes to the parser for new language constructs
4. Add new methods corresponding to the new grammar rules to our recursive-descent parser and update any existing methods, if necessary (*factor* method, I'm looking at you. :)
5. Add new visitor methods to the interpreter
6. Add a dictionary for storing variables and for looking them up

In this part I had to introduce a number of "hacks" that we'll remove as we move forward with the series:



1. The *program* grammar rule is incomplete. We'll extend it later with additional elements.
2. Pascal is a statically typed language, and you must declare a variable and its type before using it. But, as you saw, that was not the case in this article.
3. No type checking so far. It's not a big deal at this point, but I just wanted to mention it explicitly. Once we add more types to our interpreter we'll need to report an error when you try to add a string and an integer, for example.
4. A symbol table in this part is a simple Python dictionary that does double duty as a memory space. Worry not: symbol tables are such an important topic that I'll have several articles dedicated just to them. And memory space (runtime management) is a topic of its own.
5. In our simple calculator from previous articles, we used a forward slash character '/' for denoting integer division. In Pascal, though, you have to use a keyword *div* to specify integer division (See Exercise 1).
6. There is also one hack that I introduced on purpose so that you could fix it in Exercise 2: in Pascal all reserved keywords and identifiers are case insensitive, but the interpreter in this article treats them as case sensitive.

To keep you fit, here are new exercises for you:



1. Pascal variables and reserved keywords are case insensitive, unlike in many other programming languages, so *BEGIN*, *begin*, and *BeGin* they all refer to the same reserved keyword. Update the interpreter so that variables and reserved keywords are case insensitive. Use the following program to test it:

```
BEGIN

    BEGIN
        number := 2;
        a := NumBer;
        B := 10 * a + 10 * NUMBER / 4;
        c := a - - b
    end;

    x := 11;
END.
```

2. I mentioned in the "hacks" section before that our interpreter is using the forward slash character '/' to denote integer division, but instead it should be using Pascal's reserved keyword *div* for integer division. Update the interpreter to use the *div* keyword for integer division, thus eliminating one of the hacks.
3. Update the interpreter so that variables could also start with an underscore as in '_num := 5'.

That's all for today. Stay tuned and see you soon.

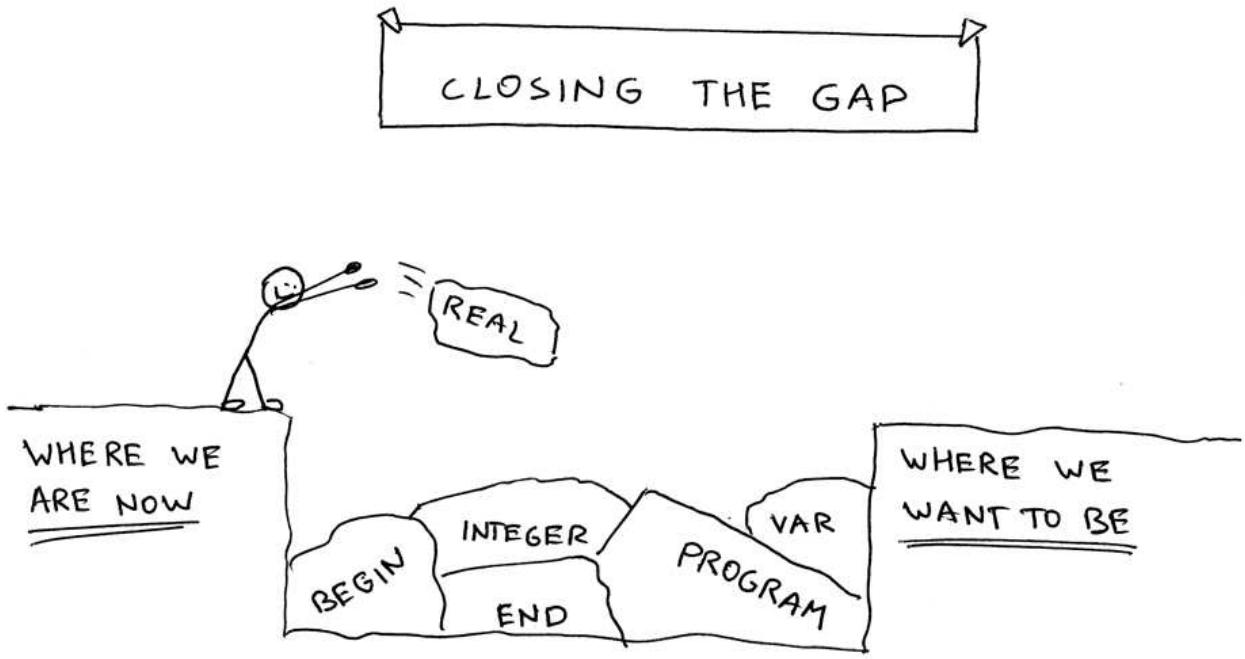
Here is a list of books I recommend that will help you in your study of interpreters and compilers:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?

Let’s Build A Simple Interpreter. Part 10. (https://ruslanspivak.com/lsbasi-part10/)

Date Thu, August 04, 2016

Today we will continue closing the gap between where we are right now and where we want to be: a fully functional interpreter for a subset of Pascal programming language (/lsbasi-part1/).



In this article we will update our interpreter to parse and interpret our very first complete Pascal program. The program can also be compiled by the Free Pascal compiler, *fpc* (http://www.freepascal.org/).

Here is the program itself:

```
PROGRAM Part10;
VAR
   number     : INTEGER;
   a, b, c, x : INTEGER;
   y          : REAL;

BEGIN {Part10}
   BEGIN
      number := 2;
      a := number;
      b := 10 * a + 10 * number DIV 4;
      c := a - - b
   END;
   x := 11;
   y := 20 / 7 + 3.14;
   { writeln('a = ', a); }
   { writeln('b = ', b); }
   { writeln('c = ', c); }
   { writeln('number = ', number); }
   { writeln('x = ', x); }
   { writeln('y = ', y); }
END.  {Part10}
```

Before we start digging into the details, download the source code of the interpreter from GitHub (https://github.com/rspivak/lsbasi/blob/master/part10/python/spi.py) and the Pascal source code above (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas), and try it on the command line:

```
$ python spi.py part10.pas
a = 2
b = 25
c = 27
number = 2
x = 11
y = 5.99714285714
```

If I remove the comments around the *writeln* statements in the part10.pas (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas) file, compile the source code with *fpc* (http://www.freepascal.org/) and then run the produced executable, this is what I get on my laptop:

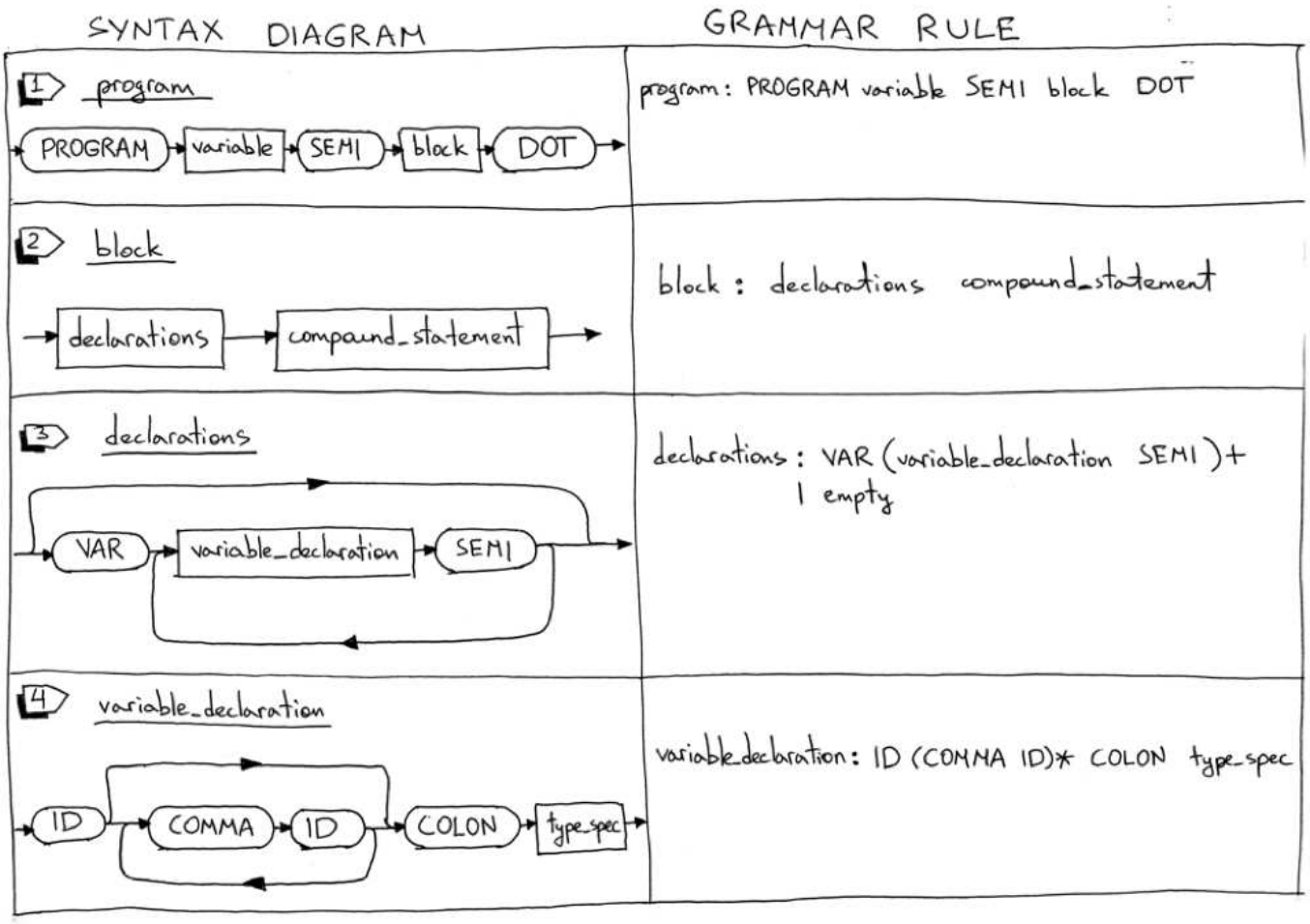
```
$ fpc part10.pas
$ ./part10
a = 2
b = 25
c = 27
number = 2
x = 11
y =  5.99714285714286E+000
```

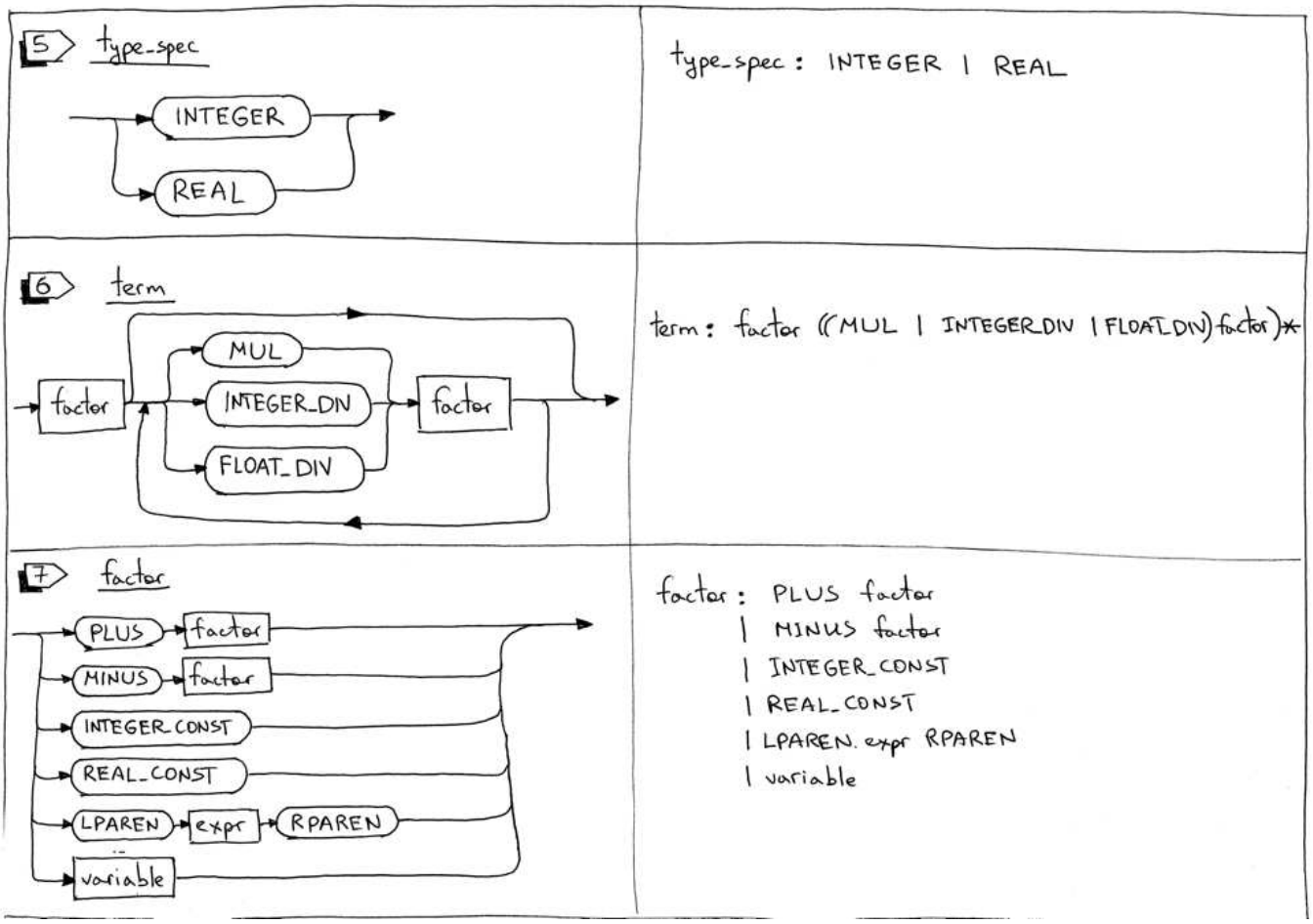
Okay, let's see what we're going cover today:

1. We will learn how to parse and interpret the Pascal ***PROGRAM*** header

2. We will learn how to parse Pascal variable declarations
3. We will update our interpreter to use the ***DIV*** keyword for integer division and a forward slash / for float division
4. We will add support for Pascal comments

Let’s dive in and look at the grammar changes first. Today we will add some new rules and update some of the existing rules.





1. The ***program*** definition grammar rule is updated to include the ***PROGRAM*** reserved keyword, the program name, and a block that ends with a dot. Here is an example of a complete Pascal program:

```
PROGRAM Part10;
BEGIN
END.
```

2. The ***block*** rule combines a *declarations* rule and a *compound_statement* rule. We'll also use the rule later in the series when we add procedure declarations. Here is an example of a block:

```
VAR
   number : INTEGER;

BEGIN
END
```

   Here is another example:

```
BEGIN
END
```

3. Pascal declarations have several parts and each part is optional. In this article, we'll cover the variable declaration part only. The ***declarations*** rule has either a variable declaration sub-rule or it's empty.

4. Pascal is a statically typed language, which means that every variable needs a variable declaration that explicitly specifies its type. In Pascal, variables must be declared before they are used. This is achieved by declaring variables in the program variable declaration section using the ***VAR*** reserved keyword. You can define variables like this:

```
VAR
   number     : INTEGER;
   a, b, c, x : INTEGER;
   y          : REAL;
```

5. The ***type_spec*** rule is for handling *INTEGER* and *REAL* types and is used in variable declarations. In the example below

```
VAR
   a : INTEGER;
   b : REAL;
```

   the variable "a" is declared with the type *INTEGER* and the variable "b" is declared with the type *REAL* (float). In this article we won't enforce type checking, but we will add type checking later in the series.

6. The ***term*** rule is updated to use the ***DIV*** keyword for integer division and a forward slash / for float division.

   Before, dividing 20 by 7 using a forward slash would produce an INTEGER 2:

```
20 / 7 = 2
```

   Now, dividing 20 by 7 using a forward slash will produce a REAL (floating point number) 2.85714285714 :

```
20 / 7 = 2.85714285714
```

   From now on, to get an INTEGER instead of a REAL, you need to use the ***DIV*** keyword:

```
20 DIV 7 = 2
```

7. The ***factor*** rule is updated to handle both integer and real (float) constants. I also removed the INTEGER sub-rule because the constants will be represented by ***INTEGER_CONST*** and ***REAL_CONST*** tokens and the ***INTEGER*** token will be used to represent the integer type. In the example below the lexer will generate an *INTEGER_CONST* token for 20 and 7 and a *REAL_CONST* token for 3.14 :

```
y := 20 / 7 + 3.14;
```

Here is our complete grammar for today:

```
program : PROGRAM variable SEMI block DOT

block : declarations compound_statement

declarations : VAR (variable_declaration SEMI)+
             | empty

variable_declaration : ID (COMMA ID)* COLON type_spec

type_spec : INTEGER

compound_statement : BEGIN statement_list END

statement_list : statement
               | statement SEMI statement_list

statement : compound_statement
          | assignment_statement
          | empty

assignment_statement : variable ASSIGN expr

empty :

expr : term ((PLUS | MINUS) term)*

term : factor ((MUL | INTEGER_DIV | FLOAT_DIV) factor)*

factor : PLUS factor
       | MINUS factor
       | INTEGER_CONST
       | REAL_CONST
       | LPAREN expr RPAREN
       | variable

variable: ID
```

In the rest of the article we'll go through the same drill we went through last time:

1. Update the lexer
2. Update the parser
3. Update the interpreter

Updating the Lexer

Here is a summary of the lexer changes:

1. New tokens
2. New and updated reserved keywords
3. New *skip_comments* method to handle Pascal comments
4. Rename the *integer* method and make some changes to the method itself
5. Update the *get_next_token* method to return new tokens

Let's dig into the changes mentioned above:

1. To handle a program header, variable declarations, integer and float constants as well as integer and float division, we need to add some new tokens - some of which are reserved keywords - and we also need to update the meaning of the INTEGER token to represent the integer type and not an integer constant. Here is a complete list of new and updated tokens:
   - PROGRAM (reserved keyword)
   - VAR (reserved keyword)
   - COLON (:)
   - COMMA (,)
   - INTEGER (we change it to mean integer type and not integer constant like 3 or 5)
   - REAL (for Pascal REAL type)
   - INTEGER_CONST (for example, 3 or 5)
   - REAL_CONST (for example, 3.14 and so on)
   - INTEGER_DIV for integer division (the ***DIV*** reserved keyword)
   - FLOAT_DIV for float division (forward slash /)
2. Here is the complete mapping of reserved keywords to tokens:

```
RESERVED_KEYWORDS = {
    'PROGRAM': Token('PROGRAM', 'PROGRAM'),
    'VAR': Token('VAR', 'VAR'),
    'DIV': Token('INTEGER_DIV', 'DIV'),
    'INTEGER': Token('INTEGER', 'INTEGER'),
    'REAL': Token('REAL', 'REAL'),
    'BEGIN': Token('BEGIN', 'BEGIN'),
    'END': Token('END', 'END'),
}
```

3. We're adding the *skip_comment* method to handle Pascal comments. The method is pretty basic and all it does is discard all the characters until the closing curly brace is found:

```
def skip_comment(self):
    while self.current_char != '}':
        self.advance()
    self.advance()  # the closing curly brace
```

4. We are renaming the *integer* method the *number* method. It can handle both integer constants and float constants like 3 and 3.14:

```
def number(self):
    """Return a (multidigit) integer or float consumed from the input."""
    result = ''
    while self.current_char is not None and self.current_char.isdigit():
        result += self.current_char
        self.advance()

    if self.current_char == '.':
        result += self.current_char
        self.advance()

        while (
            self.current_char is not None and
            self.current_char.isdigit()
        ):
            result += self.current_char
            self.advance()

        token = Token('REAL_CONST', float(result))
    else:
        token = Token('INTEGER_CONST', int(result))

    return token
```

5. We're also updating the *get_next_token* method to return new tokens:

```
def get_next_token(self):
    while self.current_char is not None:
        ...
        if self.current_char == '{':
            self.advance()
            self.skip_comment()
            continue
        ...
        if self.current_char.isdigit():
            return self.number()

        if self.current_char == ':':
            self.advance()
            return Token(COLON, ':')

        if self.current_char == ',':
            self.advance()
            return Token(COMMA, ',')
        ...
        if self.current_char == '/':
            self.advance()
            return Token(FLOAT_DIV, '/')
        ...
```

Updating the Parser

Now onto the parser changes.

Here is a summary of the changes:

1. New AST nodes: *Program*, *Block*, *VarDecl*, *Type*
2. New methods corresponding to new grammar rules: *block*, *declarations*, *variable_declaration*, and *type_spec*.
3. Updates to the existing parser methods: *program*, *term*, and *factor*

Let's go over the changes one by one:

1. We'll start with new AST nodes first. There are four new nodes:
   - The *Program* AST node represents a program and will be our root node

     ```
     class Program(AST):
         def __init__(self, name, block):
             self.name = name
             self.block = block
     ```

   - The *Block* AST node holds declarations and a compound statement:

     ```
     class Block(AST):
         def __init__(self, declarations, compound_statement):
             self.declarations = declarations
             self.compound_statement = compound_statement
     ```

   - The *VarDecl* AST node represents a variable declaration. It holds a variable node and a type node:

     ```
     class VarDecl(AST):
         def __init__(self, var_node, type_node):
             self.var_node = var_node
             self.type_node = type_node
     ```

   - The *Type* AST node represents a variable type (INTEGER or REAL):

     ```
     class Type(AST):
         def __init__(self, token):
             self.token = token
             self.value = token.value
     ```

2. As you probably remember, each rule from the grammar has a corresponding method in our recursive-descent parser. Today we're adding four new methods: *block*, *declarations*, *variable_declaration*, and *type_spec*. These methods are responsible for parsing new language constructs and constructing new AST nodes:

```
def block(self):
    """block : declarations compound_statement"""
    declaration_nodes = self.declarations()
    compound_statement_node = self.compound_statement()
    node = Block(declaration_nodes, compound_statement_node)
    return node

def declarations(self):
    """declarations : VAR (variable_declaration SEMI)+
                    | empty
    """
    declarations = []
    if self.current_token.type == VAR:
        self.eat(VAR)
        while self.current_token.type == ID:
            var_decl = self.variable_declaration()
            declarations.extend(var_decl)
            self.eat(SEMI)

    return declarations

def variable_declaration(self):
    """variable_declaration : ID (COMMA ID)* COLON type_spec"""
    var_nodes = [Var(self.current_token)]  # first ID
    self.eat(ID)

    while self.current_token.type == COMMA:
        self.eat(COMMA)
        var_nodes.append(Var(self.current_token))
        self.eat(ID)

    self.eat(COLON)

    type_node = self.type_spec()
    var_declarations = [
        VarDecl(var_node, type_node)
        for var_node in var_nodes
    ]
    return var_declarations

def type_spec(self):
    """type_spec : INTEGER
                 | REAL
    """
    token = self.current_token
    if self.current_token.type == INTEGER:
        self.eat(INTEGER)
    else:
        self.eat(REAL)
    node = Type(token)
    return node
```

3. We also need to update the *program*, *term*, and, *factor* methods to accommodate our grammar changes:

```
def program(self):
    """program : PROGRAM variable SEMI block DOT"""
    self.eat(PROGRAM)
    var_node = self.variable()
    prog_name = var_node.value
    self.eat(SEMI)
    block_node = self.block()
    program_node = Program(prog_name, block_node)
    self.eat(DOT)
    return program_node

def term(self):
    """term : factor ((MUL | INTEGER_DIV | FLOAT_DIV) factor)*"""
    node = self.factor()

    while self.current_token.type in (MUL, INTEGER_DIV, FLOAT_DIV):
        token = self.current_token
        if token.type == MUL:
            self.eat(MUL)
        elif token.type == INTEGER_DIV:
            self.eat(INTEGER_DIV)
        elif token.type == FLOAT_DIV:
            self.eat(FLOAT_DIV)

        node = BinOp(left=node, op=token, right=self.factor())

    return node

def factor(self):
    """factor : PLUS factor
              | MINUS factor
              | INTEGER_CONST
              | REAL_CONST
              | LPAREN expr RPAREN
              | variable
    """
    token = self.current_token
    if token.type == PLUS:
        self.eat(PLUS)
        node = UnaryOp(token, self.factor())
        return node
    elif token.type == MINUS:
        self.eat(MINUS)
        node = UnaryOp(token, self.factor())
        return node
    elif token.type == INTEGER_CONST:
        self.eat(INTEGER_CONST)
        return Num(token)
    elif token.type == REAL_CONST:
        self.eat(REAL_CONST)
        return Num(token)
    elif token.type == LPAREN:
        self.eat(LPAREN)
        node = self.expr()
        self.eat(RPAREN)
        return node
```

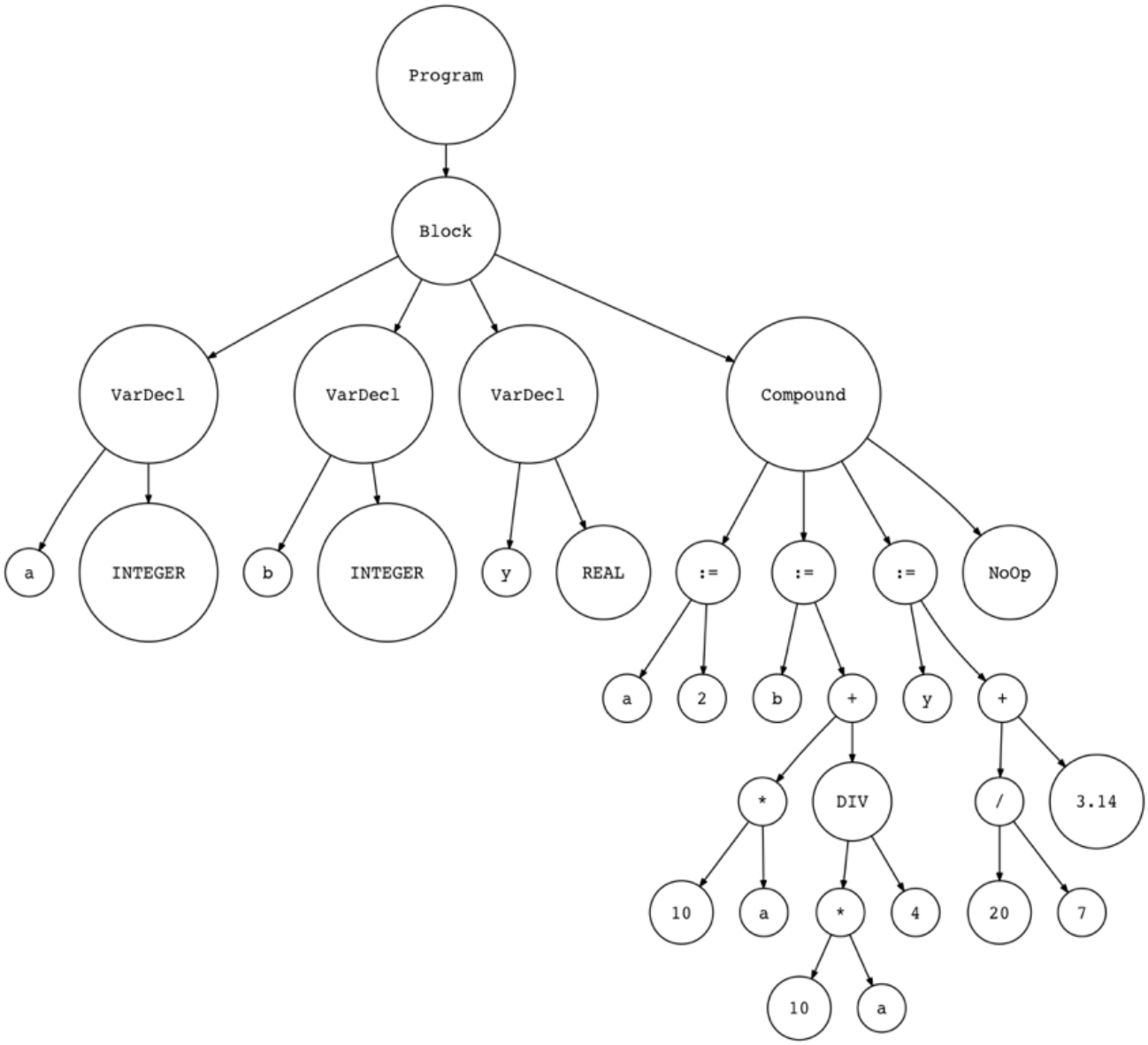
```
        else:
            node = self.variable()
            return node
```

Now, let's see what the ***Abstract Syntax Tree*** looks like with the new nodes. Here is a small working Pascal program:

```
PROGRAM Part10AST;
VAR
   a, b : INTEGER;
   y    : REAL;

BEGIN {Part10AST}
   a := 2;
   b := 10 * a + 10 * a DIV 4;
   y := 20 / 7 + 3.14;
END.  {Part10AST}
```

Let's generate an AST and visualize it with the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part10/python/genastdot.py):

```
$ python genastdot.py part10ast.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```



In the picture you can see the new nodes that we have added.

Updating the Interpreter

We're done with the lexer and parser changes. What's left is to add new visitor methods to our *Interpreter* class. There will be four new methods to visit our new nodes:

- *visit_Program*
- *visit_Block*
- *visit_VarDecl*
- *visit_Type*

They are pretty straightforward. You can also see that the *Interpreter* does nothing with *VarDecl* and *Type* nodes:

```
def visit_Program(self, node):
    self.visit(node.block)

def visit_Block(self, node):
    for declaration in node.declarations:
        self.visit(declaration)
    self.visit(node.compound_statement)

def visit_VarDecl(self, node):
    # Do nothing
    pass

def visit_Type(self, node):
    # Do nothing
    pass
```

We also need to update the *visit_BinOp* method to properly interpret integer and float divisions:

```
def visit_BinOp(self, node):
    if node.op.type == PLUS:
        return self.visit(node.left) + self.visit(node.right)
    elif node.op.type == MINUS:
        return self.visit(node.left) - self.visit(node.right)
    elif node.op.type == MUL:
        return self.visit(node.left) * self.visit(node.right)
    elif node.op.type == INTEGER_DIV:
        return self.visit(node.left) // self.visit(node.right)
    elif node.op.type == FLOAT_DIV:
        return float(self.visit(node.left)) / float(self.visit(node.right))
```

Let's sum up what we had to do to extend the Pascal interpreter in this article:

- Add new rules to the grammar and update some existing rules
- Add new tokens and supporting methods to the lexer, update and modify some existing methods
- Add new AST nodes to the parser for new language constructs
- Add new methods corresponding to the new grammar rules to our recursive-descent parser and update some existing methods
- Add new visitor methods to the interpreter and update one existing visitor method

As a result of our changes we also got rid of some of the hacks I introduced in Part 9 (/lsbasi-part9/), namely:

- Our interpreter can now handle the ***PROGRAM*** header
- Variables can now be declared using the ***VAR*** keyword
- The ***DIV*** keyword is used for integer division and a forward slash / is used for float division

If you haven't done so yet, then, as an exercise, re-implement the interpreter in this article without looking at the source code and use part10.pas (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas) as your test input file.

That's all for today. In the next article, I'll talk in greater detail about symbol table management. Stay tuned and see you soon!

By the way, I'm writing a book **"Let's Build A Web Server: First Steps"** that explains how to write a basic web server from scratch. You can get a feel for the book here (https://ruslanspivak.com/lsbaws-part1/), here (https://ruslanspivak.com/lsbaws-part2/), and here (https://ruslanspivak.com/lsbaws-part3/). Subscribe to the mailing list to get the latest updates about the book and the release date.

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series:

- Let's Build A Simple Interpreter. Part 1. (/lsbasi-part1/)
- Let's Build A Simple Interpreter. Part 2. (/lsbasi-part2/)
- Let's Build A Simple Interpreter. Part 3. (/lsbasi-part3/)
- Let's Build A Simple Interpreter. Part 4. (/lsbasi-part4/)
- Let's Build A Simple Interpreter. Part 5. (/lsbasi-part5/)
- Let's Build A Simple Interpreter. Part 6. (/lsbasi-part6/)
- Let's Build A Simple Interpreter. Part 7. (/lsbasi-part7/)
- Let's Build A Simple Interpreter. Part 8. (/lsbasi-part8/)
- Let's Build A Simple Interpreter. Part 9. (/lsbasi-part9/)
- Let's Build A Simple Interpreter. Part 10. (/lsbasi-part10/)
- Let's Build A Simple Interpreter. Part 11. (/lsbasi-part11/)
- Let's Build A Simple Interpreter. Part 12. (/lsbasi-part12/)
- Let's Build A Simple Interpreter. Part 13. (/lsbasi-part13/)
- Let's Build A Simple Interpreter. Part 14. (/lsbasi-part14/)

Comments

Let's Build A Simple Interpreter. Part 11. (https://ruslanspivak.com/lsbasi-part11/)

Date Tue, September 20, 2016

I was sitting in my room the other day and thinking about how much we had covered, and I thought I would recap what we've learned so far and what lies ahead of us.

Up until now we've learned:

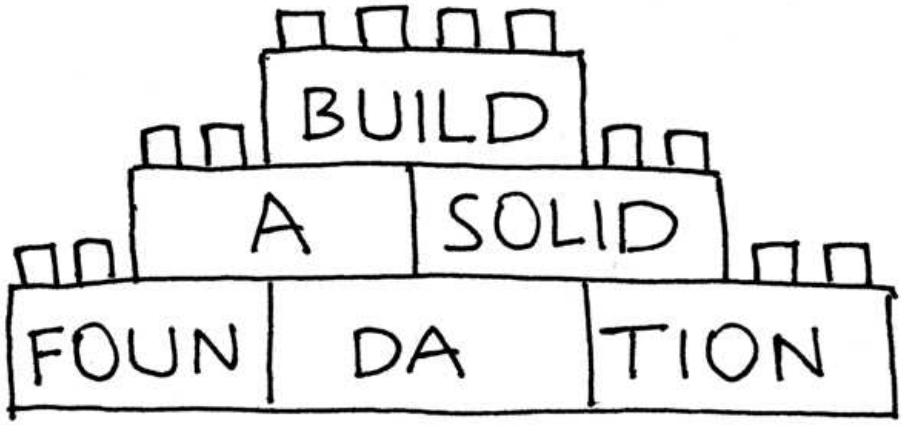
- How to break sentences into tokens. The process is called ***lexical analysis*** and the part of the interpreter that does it is called a ***lexical analyzer***, ***lexer***, ***scanner***, or ***tokenizer***. We've learned how to write our own ***lexer*** from the ground up without using regular expressions or any other tools like Lex (https://en.wikipedia.org/wiki/Lex_(software)).
- How to recognize a phrase in the stream of tokens. The process of recognizing a phrase in the stream of tokens or, to put it differently, the process of finding structure in the stream of tokens is called ***parsing*** or ***syntax analysis***. The part of an interpreter or compiler that performs that job is called a ***parser*** or ***syntax analyzer***.
- How to represent a programming language's syntax rules with ***syntax diagrams***, which are a graphical representation of a programming language's syntax rules. ***Syntax diagrams*** visually show us which statements are allowed in our programming language and which are not.
- How to use another widely used notation for specifying the syntax of a programming language. It's called ***context-free grammars*** (***grammars***, for short) or ***BNF*** (Backus-Naur Form).
- How to map a ***grammar*** to code and how to write a ***recursive-descent parser***.
- How to write a really basic ***interpreter***.
- How ***associativity*** and ***precedence*** of operators work and how to construct a grammar using a precedence table.
- How to build an ***Abstract Syntax Tree*** (AST) of a parsed sentence and how to represent the whole source program in Pascal as one big ***AST***.
- How to walk an AST and how to implement our interpreter as an AST node visitor.

With all that knowledge and experience under our belt, we've built an interpreter that can scan, parse, and build an AST and interpret, by walking the AST, our very first complete Pascal program. Ladies and gentlemen, I honestly think if you've reached this far, you deserve a pat on the back. But don't let it go to your head. Keep going. Even though we've covered a lot of ground, there are even more exciting parts coming our way.

With everything we've covered so far, we are almost ready to tackle topics like:

- Nested procedures and functions
- Procedure and function calls
- Semantic analysis (type checking, making sure variables are declared before they are used, and basically checking if a program makes sense)
- Control flow elements (like IF statements)
- Aggregate data types (Records)
- More built-in types
- Source-level debugger
- Miscellanea (All the other goodness not mentioned above :)

But before we cover those topics, we need to build a solid foundation and infrastructure.



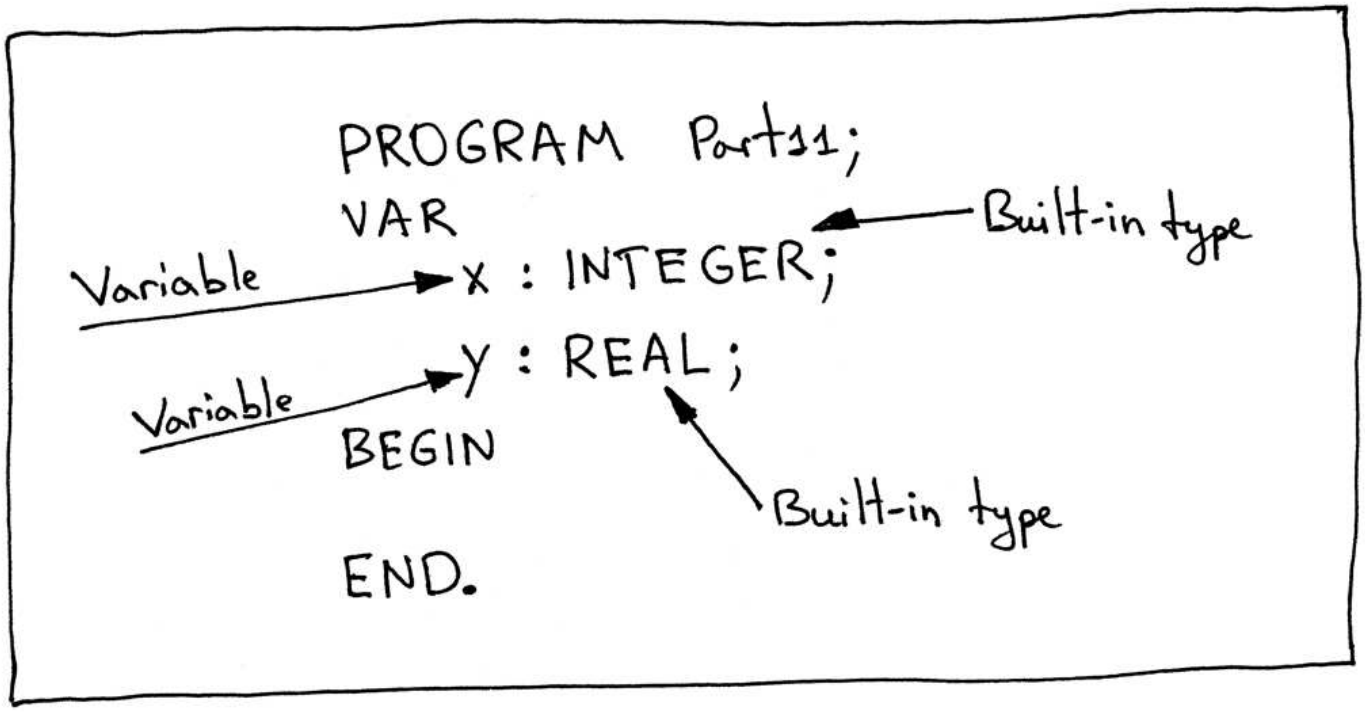
This is where we start diving deeper into the super important topic of symbols, symbol tables, and scopes. The topic itself will span several articles. It's that important and you'll see why. Okay, let's start building that foundation and infrastructure, then, shall we?

First, let's talk about symbols and why we need to track them. What is a ***symbol***? For our purposes, we'll informally define ***symbol*** as an identifier of some program entity like a variable, subroutine, or built-in type. For symbols to be useful they need to have at least the following information about the program entities they identify:

- Name (for example, 'x', 'y', 'number')
- Category (Is it a variable, subroutine, or built-in type?)
- Type (INTEGER, REAL)

Today we'll tackle variable symbols and built-in type symbols because we've already used variables and types before. By the way, the "built-in" type just means a type that hasn't been defined by you and is available for you right out of the box, like INTEGER and REAL types that you've seen and used before.

Let's take a look at the following Pascal program, specifically at the variable declaration part. You can see in the picture below that there are four symbols in that section: two variable symbols (*x* and *y*) and two built-in type symbols (*INTEGER* and *REAL*).



How can we represent symbols in code? Let's create a base *Symbol* class in Python:

```
class Symbol(object):
    def __init__(self, name, type=None):
        self.name = name
        self.type = type
```

As you can see, the class takes the *name* parameter and an optional *type* parameter (not all symbols may have a type associated with them). What about the category of a symbol? We'll encode the category of a symbol in the class name itself, which means we'll create separate classes to represent different symbol categories.

Let's start with basic built-in types. We've seen two built-in types so far, when we declared variables: INTEGER and REAL. How do we represent a built-in type symbol in code? Here is one option:

```
class BuiltinTypeSymbol(Symbol):
    def __init__(self, name):
        super(BuiltinTypeSymbol, self).__init__(name)

    def __str__(self):
        return self.name

    __repr__ = __str__
```

The class inherits from the *Symbol* class and the constructor requires only a name of the type. The category is encoded in the class name, and the *type* parameter from the base class for a built-in type symbol is *None*. The double underscore or *dunder* (as in "Double UNDERscore") methods *__str__* and *__repr__* are special Python methods and we've defined them to have a nice formatted message when you print a symbol object.

Download the interpreter file (https://github.com/rspivak/lsbasi/blob/master/part11/python/spi.py) and save it as *spi.py*; launch a python shell from the same directory where you saved the spi.py file, and play with the class we've just defined interactively:

```
$ python
>>> from spi import BuiltinTypeSymbol
>>> int_type = BuiltinTypeSymbol('INTEGER')
>>> int_type
INTEGER
>>> real_type = BuiltinTypeSymbol('REAL')
>>> real_type
REAL
```

How can we represent a variable symbol? Let's create a *VarSymbol* class:

```
class VarSymbol(Symbol):
    def __init__(self, name, type):
        super(VarSymbol, self).__init__(name, type)

    def __str__(self):
        return '<{name}:{type}>'.format(name=self.name, type=self.type)

    __repr__ = __str__
```

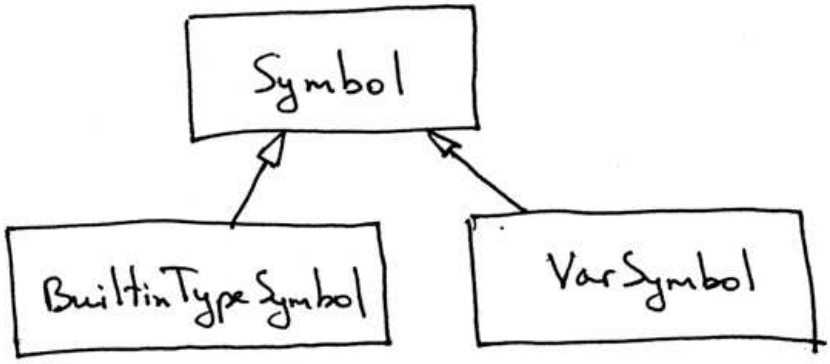
In the class we made both the *name* and the *type* parameters required parameters and the class name *VarSymbol* clearly indicates that an instance of the class will identify a variable symbol (the category is *variable*.)

Back to the interactive python shell to see how we can manually construct instances for our variable symbols now that we know how to construct *BuiltinTypeSymbol* class instances:

```
$ python
>>> from spi import BuiltinTypeSymbol, VarSymbol
>>> int_type = BuiltinTypeSymbol('INTEGER')
>>> real_type = BuiltinTypeSymbol('REAL')
>>>
>>> var_x_symbol = VarSymbol('x', int_type)
>>> var_x_symbol
<x:INTEGER>
>>> var_y_symbol = VarSymbol('y', real_type)
>>> var_y_symbol
<y:REAL>
```

As you can see, we first create an instance of a built-in type symbol and then pass it as a parameter to *VarSymbol*'s constructor.

Here is the hierarchy of symbols we've defined in visual form:

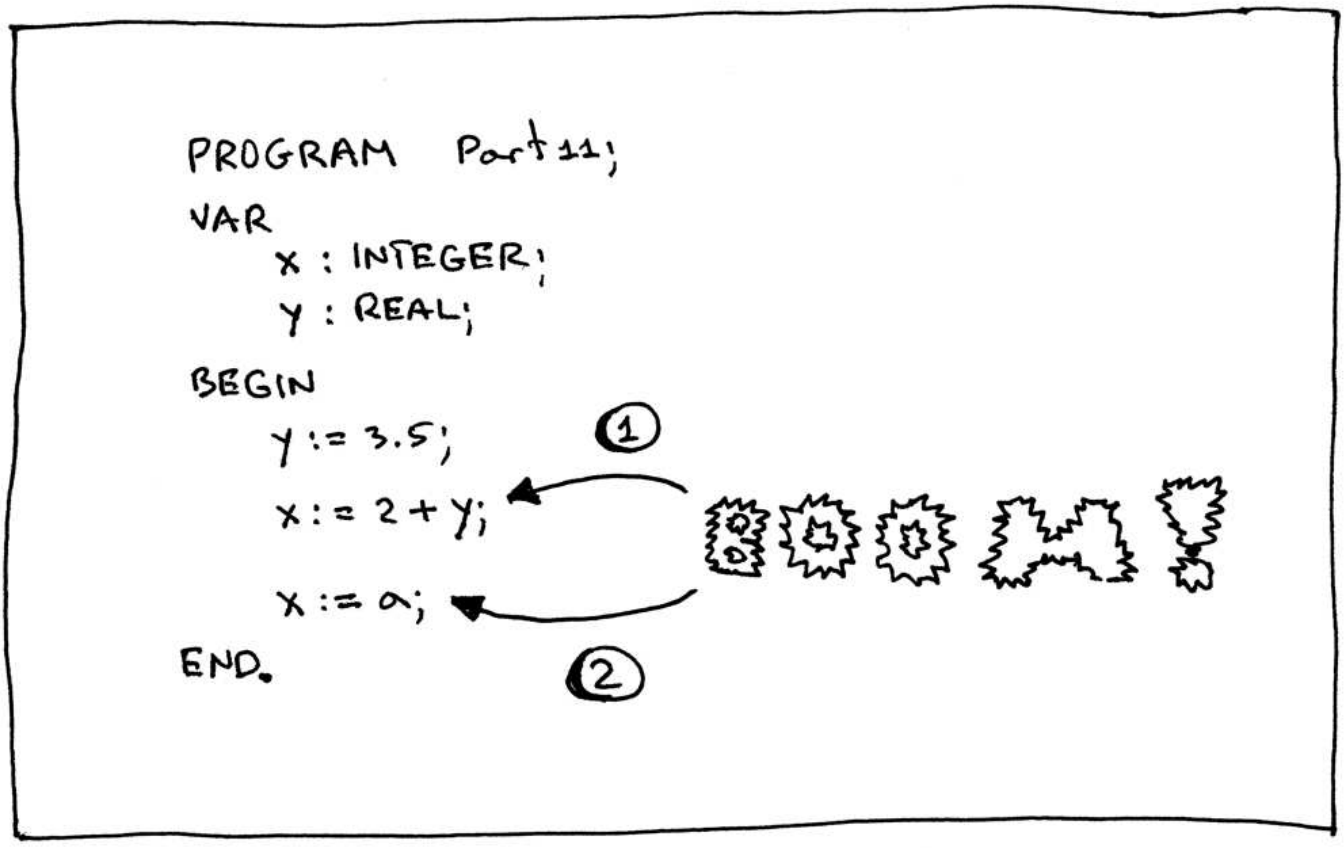


So far so good, but we haven't answered the question yet as to why we even need to track those symbols in the first place.

Here are some of the reasons:

- To make sure that when we assign a value to a variable the types are correct (type checking)
- To make sure that a variable is declared before it is used

Take a look at the following incorrect Pascal program, for example:



There are two problems with the program above (you can compile it with *fpc* (http://www.freepascal.org/) to see it for yourself):

1. In the expression *"x := 2 + y;"* we assigned a decimal value to the variable "x" that was declared as integer. That wouldn't compile because the types are incompatible.
2. In the assignment statement *"x := a;"* we referenced the variable "a" that wasn't declared - wrong!

To be able to identify cases like that even before interpreting/evaluating the source code of the program at run-time, we need to track program symbols. And where do we store the symbols that we track? I think you've guessed it right - in the symbol table!

What is a ***symbol table***? A ***symbol table*** is an abstract data type (***ADT***) for tracking various symbols in source code. Today we're going to implement our symbol table as a separate class with some helper methods:

```
class SymbolTable(object):
    def __init__(self):
        self._symbols = OrderedDict()

    def __str__(self):
        s = 'Symbols: {symbols}'.format(
            symbols=[value for value in self._symbols.values()]
        )
        return s

    __repr__ = __str__

    def define(self, symbol):
        print('Define: %s' % symbol)
        self._symbols[symbol.name] = symbol

    def lookup(self, name):
        print('Lookup: %s' % name)
        symbol = self._symbols.get(name)
        # 'symbol' is either an instance of the Symbol class or 'None'
        return symbol
```

There are two main operations that we will be performing with the symbol table: storing symbols and looking them up by name: hence, we need two helper methods - *define* and *lookup*.

The method *define* takes a symbol as a parameter and stores it internally in its *_symbols* ordered dictionary using the symbol's name as a key and the symbol instance as a value. The method *lookup* takes a symbol name as a parameter and returns a symbol if it finds it or "None" if it doesn't.

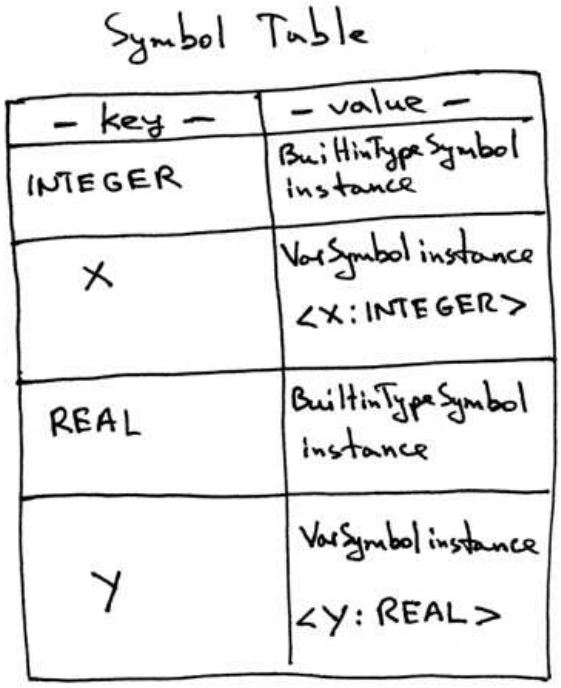
Let's manually populate our symbol table for the same Pascal program we've used just recently where we were manually creating variable and built-in type symbols:

```
PROGRAM Part11;
VAR
   x : INTEGER;
   y : REAL;

BEGIN

END.
```

Launch a Python shell again and follow along:

```
$ python
>>> from spi import SymbolTable, BuiltinTypeSymbol, VarSymbol
>>> symtab = SymbolTable()
>>> int_type = BuiltinTypeSymbol('INTEGER')
>>> symtab.define(int_type)
Define: INTEGER
>>> symtab
Symbols: [INTEGER]
>>>
>>> var_x_symbol = VarSymbol('x', int_type)
>>> symtab.define(var_x_symbol)
Define: <x:INTEGER>
>>> symtab
Symbols: [INTEGER, <x:INTEGER>]
>>>
>>> real_type = BuiltinTypeSymbol('REAL')
>>> symtab.define(real_type)
Define: REAL
>>> symtab
Symbols: [INTEGER, <x:INTEGER>, REAL]
>>>
>>> var_y_symbol = VarSymbol('y', real_type)
>>> symtab.define(var_y_symbol)
Define: <y:REAL>
>>> symtab
Symbols: [INTEGER, <x:INTEGER>, REAL, <y:REAL>]
```

If you looked at the contents of the *_symbols* dictionary it would look something like this:



How do we automate the process of building the symbol table? We'll just write another node visitor that walks the AST built by our parser! This is another example of how useful it is to have an intermediary form like AST. Instead of extending our parser to deal with the symbol table, we separate concerns and write a new node visitor class. Nice and clean. :)

Before doing that, though, let's extend our *SymbolTable* class to initialize the built-in types when the symbol table instance is created. Here is the full source code for today's *SymbolTable* class:

```
class SymbolTable(object):
    def __init__(self):
        self._symbols = OrderedDict()
        self._init_builtins()

    def _init_builtins(self):
        self.define(BuiltinTypeSymbol('INTEGER'))
        self.define(BuiltinTypeSymbol('REAL'))

    def __str__(self):
        s = 'Symbols: {symbols}'.format(
            symbols=[value for value in self._symbols.values()]
        )
        return s

    __repr__ = __str__

    def define(self, symbol):
        print('Define: %s' % symbol)
        self._symbols[symbol.name] = symbol

    def lookup(self, name):
        print('Lookup: %s' % name)
        symbol = self._symbols.get(name)
        # 'symbol' is either an instance of the Symbol class or 'None'
        return symbol
```

Now onto the *SymbolTableBuilder* AST node visitor:

```
class SymbolTableBuilder(NodeVisitor):
    def __init__(self):
        self.symtab = SymbolTable()

    def visit_Block(self, node):
        for declaration in node.declarations:
            self.visit(declaration)
        self.visit(node.compound_statement)

    def visit_Program(self, node):
        self.visit(node.block)

    def visit_BinOp(self, node):
        self.visit(node.left)
        self.visit(node.right)

    def visit_Num(self, node):
        pass

    def visit_UnaryOp(self, node):
        self.visit(node.expr)

    def visit_Compound(self, node):
        for child in node.children:
            self.visit(child)

    def visit_NoOp(self, node):
        pass

    def visit_VarDecl(self, node):
        type_name = node.type_node.value
        type_symbol = self.symtab.lookup(type_name)
        var_name = node.var_node.value
        var_symbol = VarSymbol(var_name, type_symbol)
        self.symtab.define(var_symbol)
```

You've seen most of those methods before in the *Interpreter* class, but the *visit_VarDecl* method deserves some special attention. Here it is again:

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.symtab.lookup(type_name)
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)
    self.symtab.define(var_symbol)
```

This method is responsible for visiting (walking) a *VarDecl* AST node and storing the corresponding symbol in the symbol table. First, the method looks up the built-in type symbol by name in the symbol table, then it creates an instance of the *VarSymbol* class and stores (defines) it in the symbol table.

Let's take our *SymbolTableBuilder* AST walker for a test drive and see it in action:

```
$ python
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM Part11;
... VAR
...    x : INTEGER;
...    y : REAL;
...
... BEGIN
...
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <x:INTEGER>
Lookup: REAL
Define: <y:REAL>
>>> # Let's examine the contents of our symbol table
…
>>> symtab_builder.symtab
Symbols: [INTEGER, REAL, <x:INTEGER>, <y:REAL>]
```

In the interactive session above, you can see the sequence of "Define: …" and "Lookup: …" messages that indicate the order in which symbols are defined and looked up in the symbol table. The last command in the session prints the contents of the symbol table and you can see that it's exactly the same as the contents of the symbol table that we've built manually before. The magic of AST node visitors is that they pretty much do all the work for you. :)

We can already put our symbol table and symbol table builder to good use: we can use them to verify that variables are declared before they are used in assignments and expressions. All we need to do is just extend the visitor with two more methods: *visit_Assign* and *visit_Var*:

```
def visit_Assign(self, node):
    var_name = node.left.value
    var_symbol = self.symtab.lookup(var_name)
    if var_symbol is None:
        raise NameError(repr(var_name))

    self.visit(node.right)

def visit_Var(self, node):
    var_name = node.value
    var_symbol = self.symtab.lookup(var_name)

    if var_symbol is None:
        raise NameError(repr(var_name))
```

These methods will raise a *NameError* exception if they cannot find the symbol in the symbol table.

Take a look at the following program, where we reference the variable "b" that hasn't been declared yet:

```
PROGRAM NameError1;
VAR
   a : INTEGER;

BEGIN
   a := 2 + b;
END.
```

Let's see what happens if we construct an AST for the program and pass it to our symbol table builder to visit:

```
$ python
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM NameError1;
... VAR
...    a : INTEGER;
...
... BEGIN
...    a := 2 + b;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <a:INTEGER>
Lookup: a
Lookup: b
Traceback (most recent call last):
  ...
  File "spi.py", line 674, in visit_Var
    raise NameError(repr(var_name))
NameError: 'b'
```

Exactly what we were expecting!

Here is another error case where we try to assign a value to a variable that hasn't been defined yet, in this case the variable 'a':

```
PROGRAM NameError2;
VAR
   b : INTEGER;

BEGIN
   b := 1;
   a := b + 2;
END.
```

Meanwhile, in the Python shell:

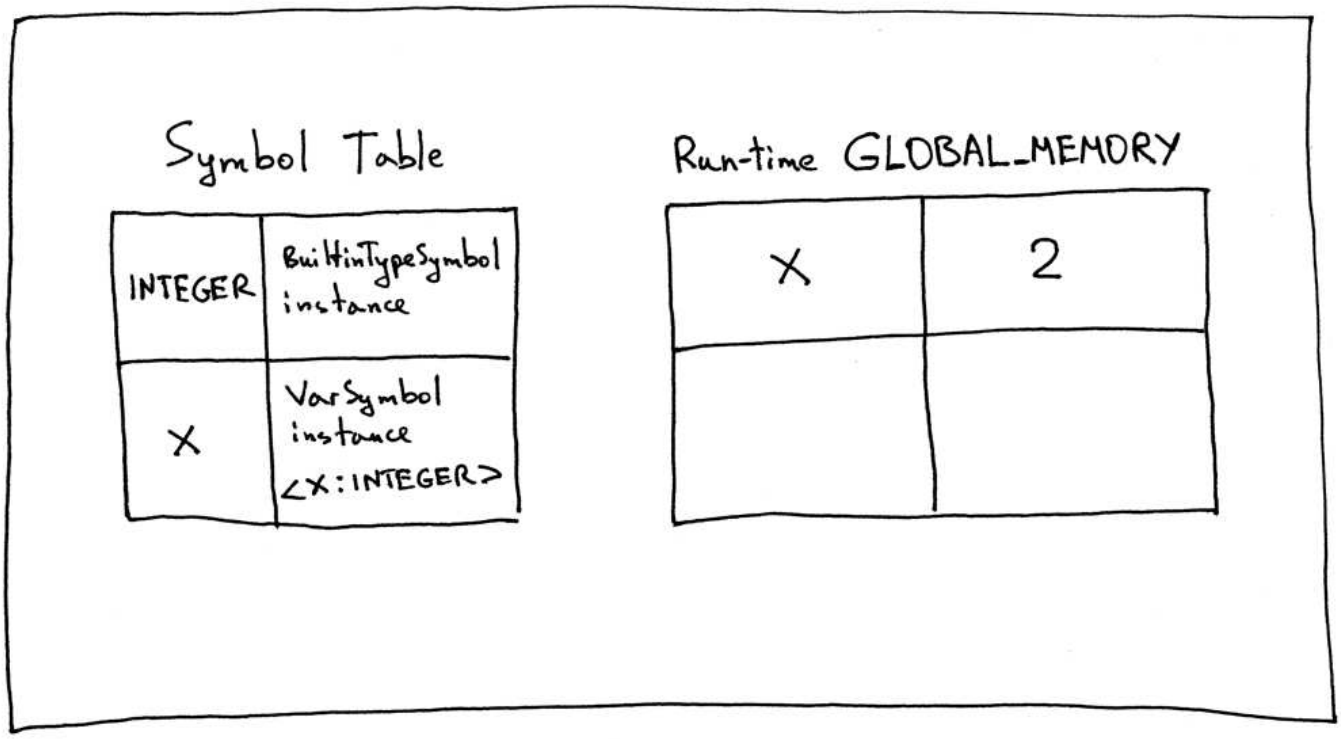
```
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM NameError2;
... VAR
...    b : INTEGER;
...
... BEGIN
...    b := 1;
...    a := b + 2;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <b:INTEGER>
Lookup: b
Lookup: a
Traceback (most recent call last):
  ...
  File "spi.py", line 665, in visit_Assign
    raise NameError(repr(var_name))
NameError: 'a'
```

Great, our new visitor caught this problem too!

I would like to emphasize the point that all those checks that our *SymbolTableBuilder* AST visitor makes are made before the run-time, so before our interpreter actually evaluates the source program. To drive the point home if we were to interpret the following program:

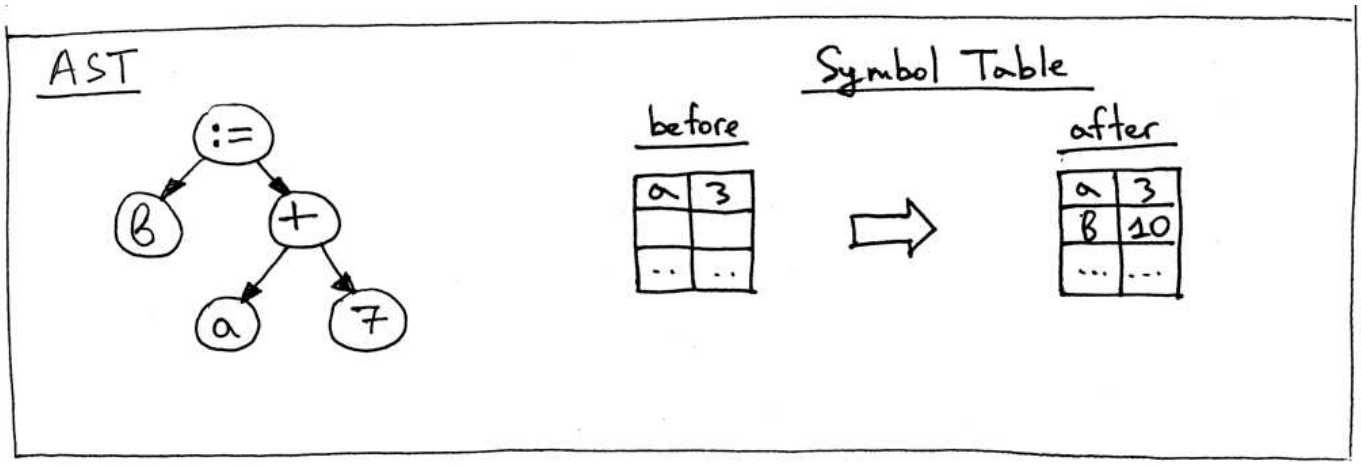
```
PROGRAM Part11;
VAR
   x : INTEGER;
BEGIN
   x := 2;
END.
```

The contents of the symbol table and the run-time GLOBAL_MEMORY right before the program exited would look something like this:



Do you see the difference? Can you see that the symbol table doesn't hold the value 2 for variable "x"? That's solely the interpreter's job now.

Remember the picture from Part 9 (/lsbasi-part9/) where the Symbol Table was used as global memory?



No more! We effectively got rid of the hack where symbol table did double duty as global memory.

Let's put it all together and test our new interpreter with the following program:

```
PROGRAM Part11;
VAR
   number : INTEGER;
   a, b   : INTEGER;
   y      : REAL;

BEGIN {Part11}
   number := 2;
   a := number ;
   b := 10 * a + 10 * number DIV 4;
   y := 20 / 7 + 3.14
END.  {Part11}
```

Save the program as part11.pas and fire up the interpreter:

```
$ python spi.py part11.pas
Define: INTEGER
Define: REAL
Lookup: INTEGER
Define: <number:INTEGER>
Lookup: INTEGER
Define: <a:INTEGER>
Lookup: INTEGER
Define: <b:INTEGER>
Lookup: REAL
Define: <y:REAL>
Lookup: number
Lookup: a
Lookup: number
Lookup: b
Lookup: a
Lookup: number
Lookup: y

Symbol Table contents:
Symbols: [INTEGER, REAL, <number:INTEGER>, <a:INTEGER>, <b:INTEGER>, <y:REAL>]

Run-time GLOBAL_MEMORY contents:
a = 2
b = 25
number = 2
y = 5.99714285714
```

I'd like to draw your attention again to the fact that the *Interpreter* class has nothing to do with building the symbol table and it relies on the *SymbolTableBuilder* to make sure that the variables in the source code are properly declared before they are used by the *Interpreter*.

Check your understanding

- What is a symbol?
- Why do we need to track symbols?
- What is a symbol table?

- What is the difference between defining a symbol and resolving/looking up the symbol?
- Given the following small Pascal program, what would be the contents of the symbol table, the global memory (the GLOBAL_MEMORY dictionary that is part of the *Interpreter*)?

```
PROGRAM Part11;
VAR
   x, y : INTEGER;
BEGIN
   x := 2;
   y := 3 + x;
END.
```

That's all for today. In the next article, I'll talk about scopes and we'll get our hands dirty with parsing nested procedures. Stay tuned and see you soon! And remember that no matter what, "Keep going!"

P.S. My explanation of the topic of symbols and symbol table management is heavily influenced by the book *Language Implementation Patterns (http://amzn.to/2cHsHT1)* by Terence Parr. It's a terrific book. I think it has the clearest explanation of the topic I've ever seen and it also covers class scopes, a subject that I'm not going to cover in the series because we will not be discussing object-oriented Pascal.

P.P.S.: If you can't wait and want to start digging into compilers, I highly recommend the freely available classic by Jack Crenshaw "Let's Build a Compiler." (http://compilers.iecc.com/crenshaw/)

By the way, I'm writing a book **"Let's Build A Web Server: First Steps"** that explains how to write a basic web server from scratch. You can get a feel for the book here (https://ruslanspivak.com/lsbaws-part1/), here (https://ruslanspivak.com/lsbaws-part2/), and here (https://ruslanspivak.com/lsbaws-part3/). Subscribe to the mailing list to get the latest updates about the book and the release date.

Enter Your First Name *

Let’s Build A Simple Interpreter. Part 12. (https://ruslanspivak.com/lsbasi-part12/)

Date Thu, December 01, 2016

> *“Be not afraid of going slowly; be afraid only of standing still.” - Chinese proverb.*

Hello, and welcome back!

Today we are going to take a few more baby steps and learn how to parse Pascal procedure declarations.

What is a ***procedure declaration***? A ***procedure declaration*** is a language construct that defines an identifier (a procedure name) and associates it with a block of Pascal code.

Before we dive in, a few words about Pascal procedures and their declarations:

- Pascal procedures don’t have return statements. They exit when they reach the end of their corresponding block.
- Pascal procedures can be nested within each other.
- For simplicity reasons, procedure declarations in this article won’t have any formal parameters. But, don’t worry, we’ll cover that later in the series.

This is our test program for today:

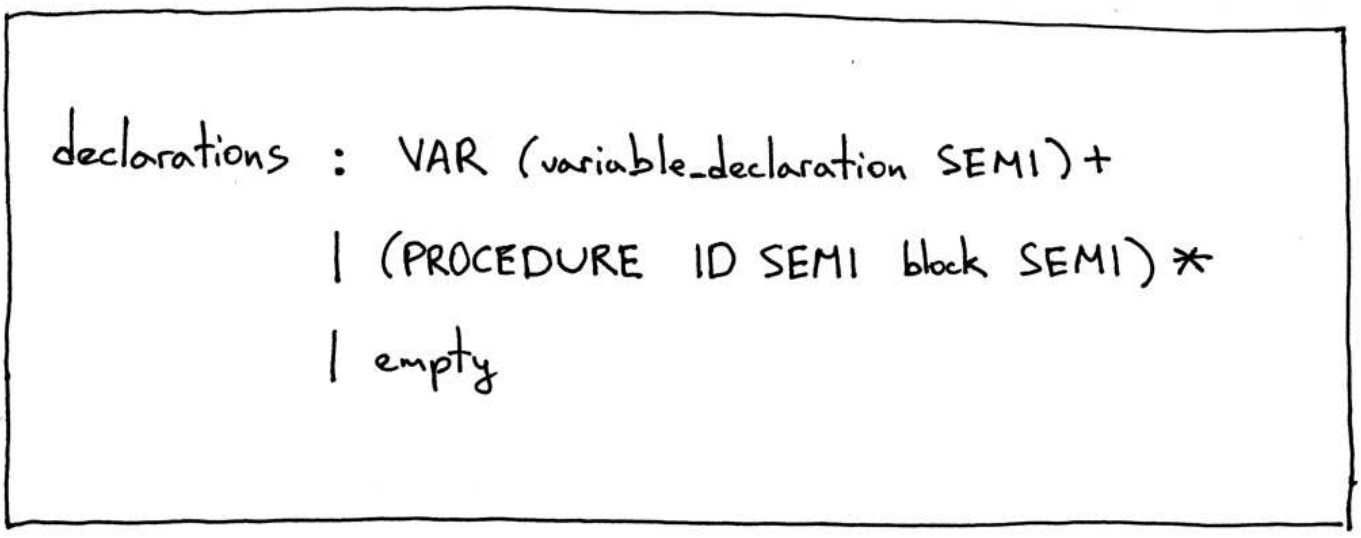
```
PROGRAM Part12;
VAR
   a : INTEGER;

PROCEDURE P1;
VAR
   a : REAL;
   k : INTEGER;

   PROCEDURE P2;
   VAR
      a, z : INTEGER;
   BEGIN {P2}
      z := 777;
   END;  {P2}

BEGIN {P1}

END;  {P1}

BEGIN {Part12}
   a := 10;
END.  {Part12}
```

As you can see above, we have defined two procedures (*P1* and *P2*) and *P2* is nested within *P1*. In the code above, I used comments with a procedure's name to clearly indicate where the body of every procedure begins and where it ends.

Our objective for today is pretty clear: learn how to parse a code like that.

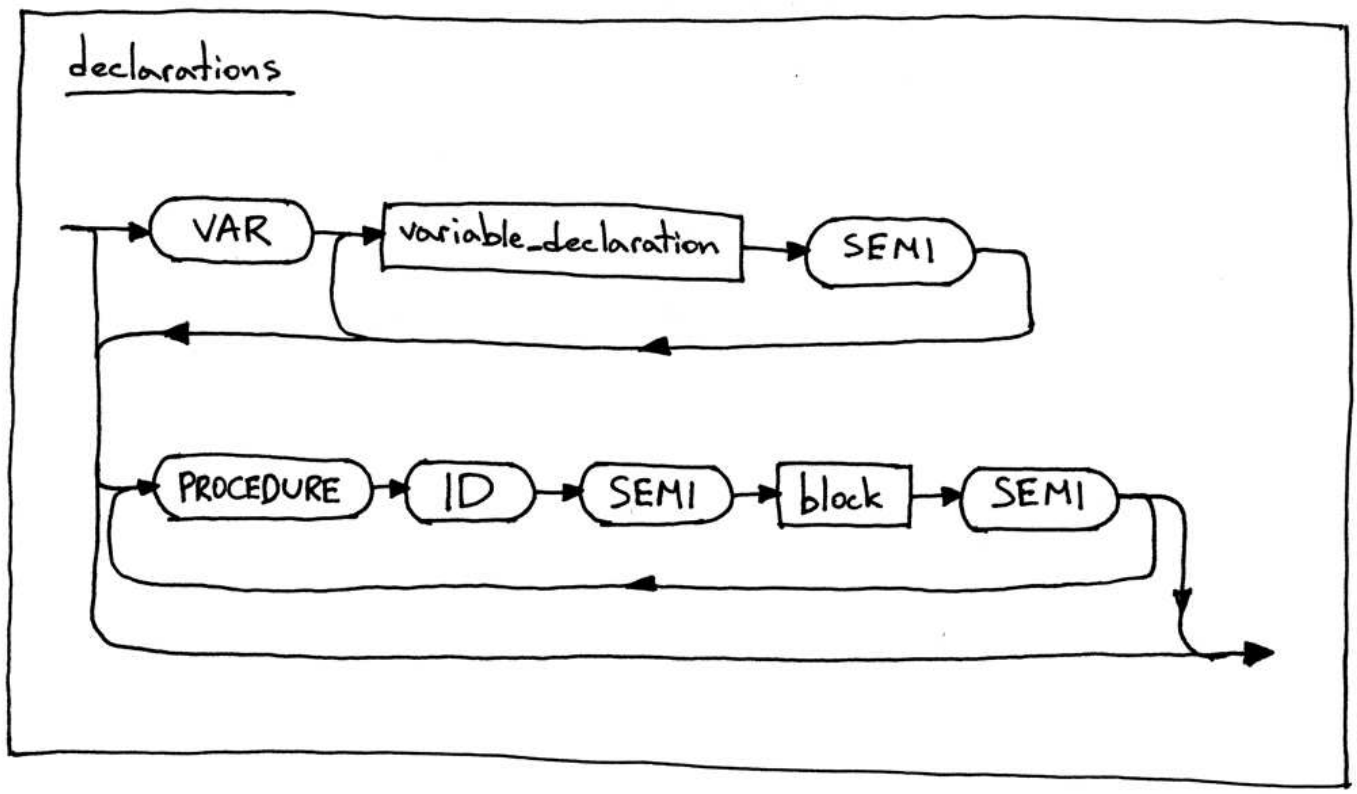
First, we need to make some changes to our grammar to add procedure declarations. Well, let's just do that!

Here is the updated *declarations* grammar rule:



The procedure declaration sub-rule consists of the reserved keyword **PROCEDURE** followed by an identifier (a procedure name), followed by a semicolon, which in turn is followed by a *block* rule, which is terminated by a semicolon. Whoa! This is a case where I think the picture is actually worth however many words I just put in the previous sentence! :)

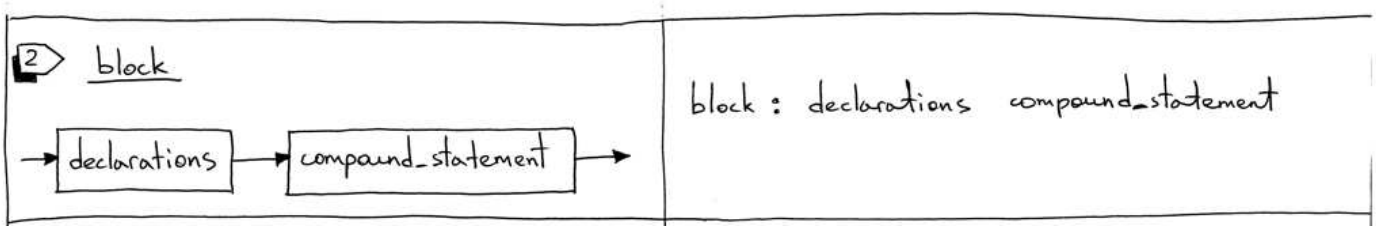
Here is the updated syntax diagram for the *declarations* rule:



From the grammar and the diagram above you can see that you can have as many procedure declarations on the same level as you want. For example, in the code snippet below we define two procedure declarations, *P1* and *P1A*, on the same level:

```
PROGRAM Test;
VAR
   a : INTEGER;

PROCEDURE P1;
BEGIN {P1}

END;  {P1}

PROCEDURE P1A;
BEGIN {P1A}

END;  {P1A}

BEGIN {Test}
   a := 10;
END.  {Test}
```

The diagram and the grammar rule above also indicate that procedure declarations can be nested because the *procedure declaration* sub-rule references the *block* rule which contains the *declarations* rule, which in turn contains the *procedure declaration* sub-rule. As a reminder, here is the syntax diagram and the grammar for the block rule from Part10 (/lsbasi-part10/):



Okay, now let's focus on the interpreter components that need to be updated to support procedure declarations:

Updating the Lexer

All we need to do is add a new token named **PROCEDURE**:

```
PROCEDURE = 'PROCEDURE'
```

And add *'PROCEDURE'* to the reserved keywords. Here is the complete mapping of reserved keywords to tokens:

```
RESERVED_KEYWORDS = {
    'PROGRAM': Token('PROGRAM', 'PROGRAM'),
    'VAR': Token('VAR', 'VAR'),
    'DIV': Token('INTEGER_DIV', 'DIV'),
    'INTEGER': Token('INTEGER', 'INTEGER'),
    'REAL': Token('REAL', 'REAL'),
    'BEGIN': Token('BEGIN', 'BEGIN'),
    'END': Token('END', 'END'),
    'PROCEDURE': Token('PROCEDURE', 'PROCEDURE'),
}
```

Updating the Parser

Here is a summary of the parser changes:

1. New *ProcedureDecl* AST node
2. Update to the parser's *declarations* method to support procedure declarations

Let's go over the changes.

1. The *ProcedureDecl* AST node represents a *procedure declaration*. The class constructor takes as parameters the name of the procedure and the AST node of the block of code that the procedure's name refers to.

   ```
   class ProcedureDecl(AST):
       def __init__(self, proc_name, block_node):
           self.proc_name = proc_name
           self.block_node = block_node
   ```

2. Here is the updated *declarations* method of the *Parser* class

```
def declarations(self):
    """declarations : VAR (variable_declaration SEMI)+
                    | (PROCEDURE ID SEMI block SEMI)*
                    | empty
    """
    declarations = []

    if self.current_token.type == VAR:
        self.eat(VAR)
        while self.current_token.type == ID:
            var_decl = self.variable_declaration()
            declarations.extend(var_decl)
            self.eat(SEMI)

    while self.current_token.type == PROCEDURE:
        self.eat(PROCEDURE)
        proc_name = self.current_token.value
        self.eat(ID)
        self.eat(SEMI)
        block_node = self.block()
        proc_decl = ProcedureDecl(proc_name, block_node)
        declarations.append(proc_decl)
        self.eat(SEMI)

    return declarations
```

Hopefully, the code above is pretty self-explanatory. It follows the grammar/syntax diagram for procedure declarations that you've seen earlier in the article.

Updating the SymbolTable builder

Because we're not ready yet to handle nested procedure scopes, we'll simply add an empty *visit_ProcedureDecl* method to the *SymbolTreeBuilder* AST visitor class. We'll fill it out in the next article.

```
def visit_ProcedureDecl(self, node):
    pass
```

Updating the Interpreter

We also need to add an empty *visit_ProcedureDecl* method to the *Interpreter* class, which will cause our interpreter to silently ignore all our procedure declarations.

So far, so good.

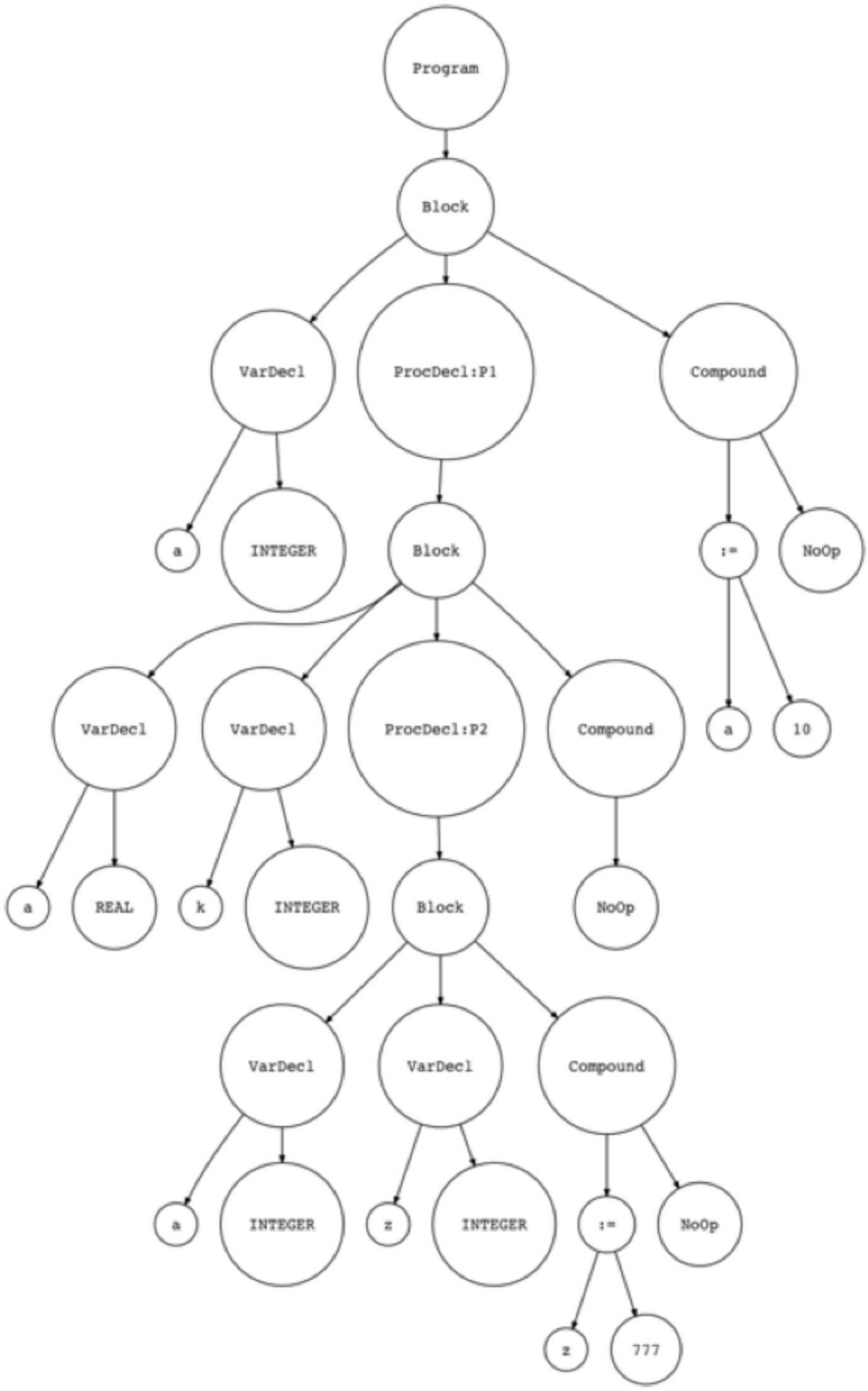
Now that we've made all the necessary changes, let's see what the *Abstract Syntax Tree* looks like with the new *ProcedureDecl* nodes.

Here is our Pascal program again (you can download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part12/python/part12.pas)):

```
PROGRAM Part12;
VAR
   a : INTEGER;

PROCEDURE P1;
VAR
   a : REAL;
   k : INTEGER;

   PROCEDURE P2;
   VAR
      a, z : INTEGER;
   BEGIN {P2}
      z := 777;
   END;  {P2}

BEGIN {P1}

END;  {P1}

BEGIN {Part12}
   a := 10;
END.  {Part12}
```

Let's generate an AST and visualize it with the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part12/python/genastdot.py) utility:

```
$ python genastdot.py part12.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```



In the picture above you can see two *ProcedureDecl* nodes: *ProcDecl:P1* and *ProcDecl:P2* that correspond to procedures *P1* and *P2*. Mission accomplished. :)

As a last item for today, let's quickly check that our updated interpreter works as before when a Pascal program has procedure declarations in it. Download the interpreter (https://github.com/rspivak/lsbasi/blob/master/part12/python/spi.py) and the test program (https://github.com/rspivak/lsbasi/blob/master/part12/python/part12.pas) if you haven't done so yet, and run it on the command line. Your output should look similar to this:

```
$ python spi.py part12.pas
Define: INTEGER
Define: REAL
Lookup: INTEGER
Define: <a:INTEGER>
Lookup: a

Symbol Table contents:
Symbols: [INTEGER, REAL, <a:INTEGER>]

Run-time GLOBAL_MEMORY contents:
a = 10
```

Okay, with all that knowledge and experience under our belt, we're ready to tackle the topic of nested scopes that we need to understand in order to be able to analyze nested procedures and prepare ourselves to handle procedure and function calls. And that's exactly what we are going to do in the next article: dive deep into nested scopes. So don't forget to bring your swimming gear next time! Stay tuned and see you soon!

All articles in this series:

- Let's Build A Simple Interpreter. Part 1. (/lsbasi-part1/)
- Let's Build A Simple Interpreter. Part 2. (/lsbasi-part2/)
- Let's Build A Simple Interpreter. Part 3. (/lsbasi-part3/)
- Let's Build A Simple Interpreter. Part 4. (/lsbasi-part4/)
- Let's Build A Simple Interpreter. Part 5. (/lsbasi-part5/)
- Let's Build A Simple Interpreter. Part 6. (/lsbasi-part6/)
- Let's Build A Simple Interpreter. Part 7. (/lsbasi-part7/)
- Let's Build A Simple Interpreter. Part 8. (/lsbasi-part8/)
- Let's Build A Simple Interpreter. Part 9. (/lsbasi-part9/)
- Let's Build A Simple Interpreter. Part 10. (/lsbasi-part10/)
- Let's Build A Simple Interpreter. Part 11. (/lsbasi-part11/)
- Let's Build A Simple Interpreter. Part 12. (/lsbasi-part12/)
- Let's Build A Simple Interpreter. Part 13. (/lsbasi-part13/)
- Let's Build A Simple Interpreter. Part 14. (/lsbasi-part14/)

Comments

John Lee • a year ago

I think I might be missing something? Shouldn't the BNF for the declarations block now be:

```
declarations : VAR (variable_declaration SEMI)+ (procedure_declaration)*
```

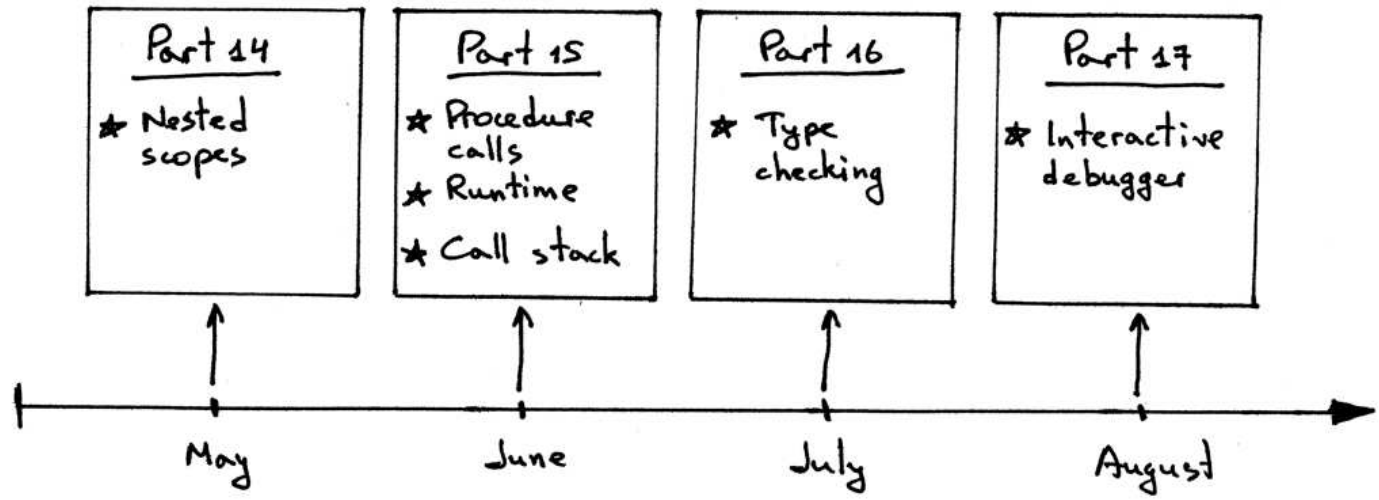
Let’s Build A Simple Interpreter. Part 13: Semantic Analysis. (https://ruslanspivak.com/lsbasi-part13/)

Date Thu, April 27, 2017

> *Anything worth doing is worth overdoing.*

Before doing a deep dive into the topic of scopes, I’d like to make a “quick” detour and talk in more detail about symbols, symbol tables, and semantic analysis. In the spirit of *“Anything worth doing is worth overdoing”*, I hope you’ll find the material useful for building a more solid foundation before tackling nested scopes. Today we will continue to increase our knowledge of how to write interpreters and compilers. You will see that some of the material covered in this article has parts that are much more extended versions of what you saw in Part 11 (/lsbasi-part11/), where we discussed symbols and symbol tables.

In the final four articles of the series we’ll discuss the remaining bits and pieces. Below you can see the major topics we will cover and the timeline:



Okay, let's get started!

Introduction to semantic analysis

While our Pascal program can be grammatically correct and the parser can successfully build an *abstract syntax tree*, the program still can contain some pretty serious errors. To catch those errors we need to use the *abstract syntax tree* and the information from the *symbol table*.

Why can't we check for those errors during parsing, that is, during *syntax analysis*? Why do we have to build an *AST* and something called the symbol table to do that?

In a nutshell, for convenience and the separation of concerns. By moving those extra checks into a separate phase, we can focus on one task at a time without making our parser and interpreter do more work than they are supposed to do.

When the parser has finished building the AST, we know that the program is grammatically correct; that is, that its syntax is correct according to our grammar rules and now we can separately focus on checking for errors that require additional context and information that the parser did not have at the time of building the AST. To make it more concrete, let's take a look at the following Pascal assignment statement:

```
x := x + y;
```

The parser will handle it all right because, grammatically, the statement is correct (according to our previously defined grammar rules for assignment statements and expressions). But that's not the end of the story yet, because Pascal has a requirement that variables must be declared with their corresponding types before they are used. How does the parser know whether **x** and **y** have been declared yet?

Well, it doesn't and that's why we need a separate semantic analysis phase to answer the question (among many others) of whether the variables have been declared prior to their use.

What is ***semantic analysis***? Basically, it's just a process to help us determine whether a program makes sense, and that it has meaning, according to a language definition.

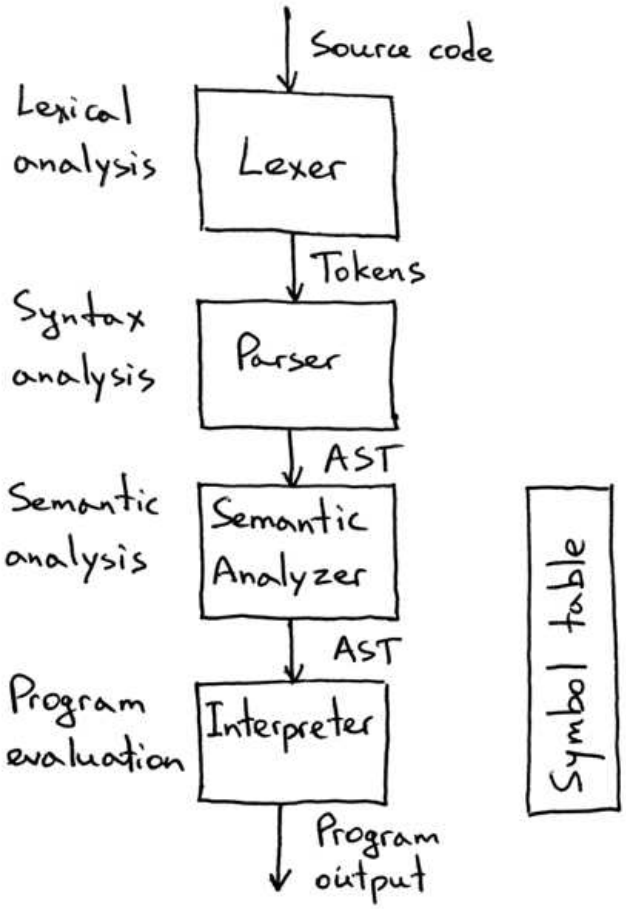
What does it even mean for a program to make sense? It depends in large part on a language definition and language requirements.

Pascal language and, specifically, Free Pascal's compiler, has certain requirements that, if not followed in a program, would lead to an error from the *fpc* compiler indicating that the program doesn't "make sense", that it is incorrect, even though the syntax might look okay. Here are some of those requirements:

- The variables must be declared before they are used
- The variables must have matching types when used in arithmetic expressions (this is a big part of *semantic analysis* called *type checking* that we'll cover separately)
- There should be no duplicate declarations (Pascal prohibits, for example, having a local variable in a procedure with the same name as one of the procedure's formal parameters)
- A name reference in a call to a procedure must refer to the actual declared procedure (It doesn't make sense in Pascal if, in the procedure call **foo()**, the name *foo* refers to a variable foo of a primitive type INTEGER)
- A procedure call must have the correct number of arguments and the arguments' types must match those of formal parameters in the procedure declaration

It is much easier to enforce the above requirements when we have enough context about the program, namely, an intermediate representation in the form of an AST that we can walk and the symbol table with information about different program entities like variables, procedures, and functions.

After we implement the semantic analysis phase, the structure of our Pascal interpreter will look something like this:



From the picture above you can see that our lexer will get source code as an input, transform that into tokens that the parser will consume and use to verify that the program is grammatically correct, and then it will generate an abstract syntax tree that our new semantic analysis phase will use to enforce

different Pascal language requirements. During the semantic analysis phase, the semantic analyzer will also build and use the symbol table. After the semantic analysis, our interpreter will take the AST, evaluate the program by walking the AST, and produce the program output.

Let's get into the details of the semantic analysis phase.

Symbols and symbol tables

In the following section, we're going to discuss how to implement some of the semantic checks and how to build the symbol table: in other words, we are going to discuss how to perform a *semantic analysis* of our Pascal programs. Keep in mind that even though *semantic analysis* sounds fancy and deep, it's just another step after parsing our program and creating an AST to check the source program for some additional errors that the parser couldn't catch due to a lack of additional information (context).

Today we're going to focus on the following two *static semantic checks**:

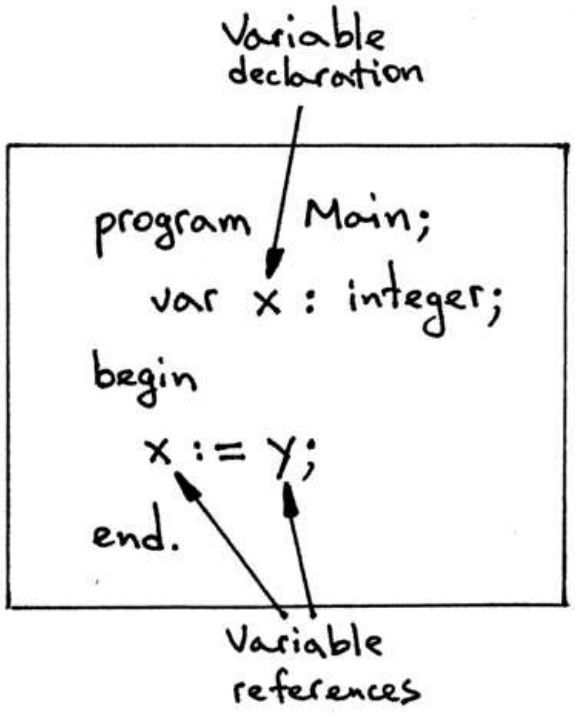
1. That variables are declared before they are used
2. That there are no duplicate variable declarations

> *ASIDE: *Static semantic checks* are the checks that we can make before interpreting (evaluating) the program, that is, before calling the interpret method on an instance of the Interpreter class. All the Pascal requirements mentioned before can be enforced with *static semantic checks* by walking an AST and using information from the symbol table.
>
> *Dynamic semantic checks*, on the other hand, would require checks to be performed during the interpretation (evaluation) of the program. For example, a check that there is no division by zero, and that an array index is not out of bounds would be a *dynamic semantic check*. Our focus today is on *static semantic checks*.

Let's start with our first check and make sure that in our Pascal programs variables are declared before they are used. Take a look at the following syntactically correct but semantically incorrect program (ugh… too many hard to pronounce words in one sentence. :)

```
program Main;
   var x : integer;

begin
    x := y;
end.
```

The program above has one variable declaration and two variable references. You can see that in the picture below:



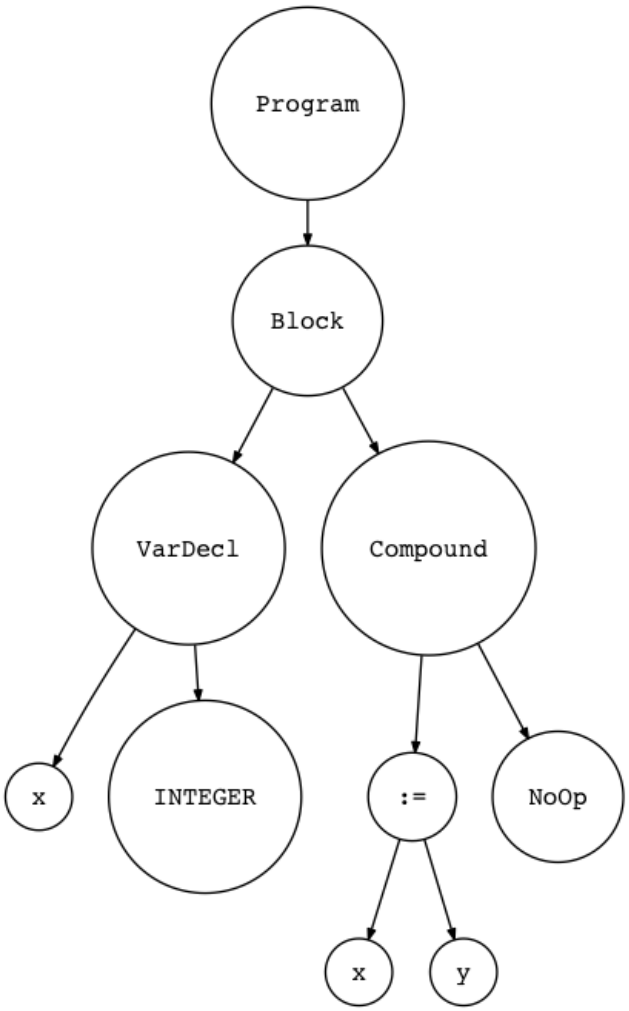
Let's actually verify that our program is syntactically correct and that our parser doesn't throw an error when parsing it. As they say, trust but verify. :) Download spi.py (https://github.com/rspivak/lsbasi/blob/master/part13/spi.py), fire off a Python shell, and see for yourself:

```
>>> from spi import Lexer, Parser
>>> text = """
program Main;
   var x : integer;

begin
    x := y;
end.
"""
>>>
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>>
```

You see? No errors. We can even generate an AST diagram for that program using genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part13/genastdot.py). First, save the source code into a file, let's say semanticerror01.pas, and run the following commands:

```
$ python genastdot.py semanticerror01.pas > semanticerror01.dot
$ dot -Tpng -o ast.png semanticerror01.dot
```

Here is the AST diagram:



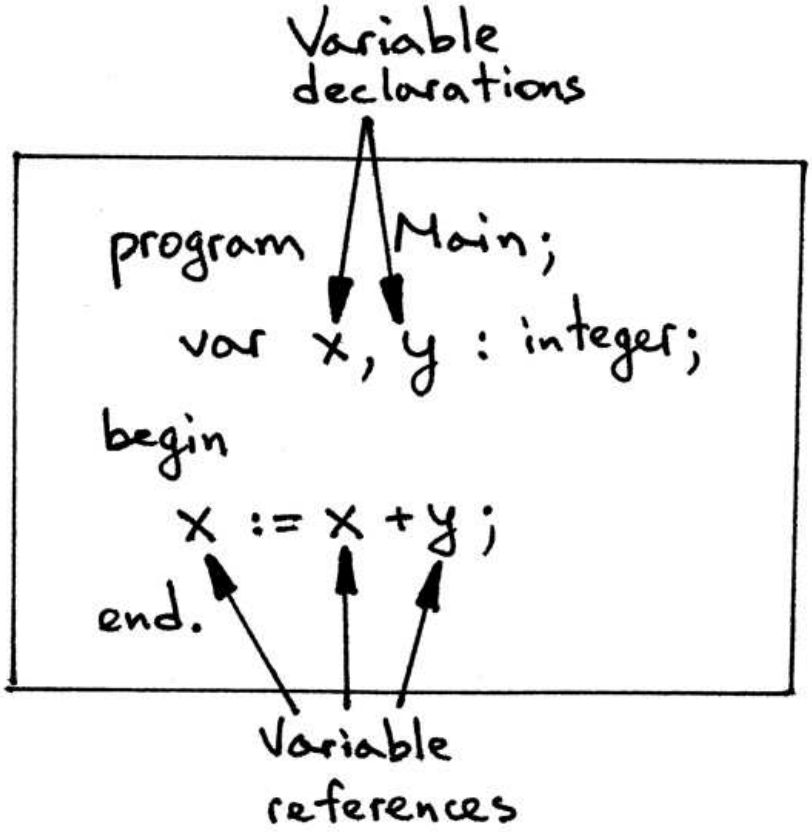
So, it is a grammatically (syntactically) correct program, but the program doesn't make sense because we don't even know what type the variable **y** has (that's why we need declarations) and if it will make sense to assign **y** to **x**. What if **y** is a string, does it make sense to assign a string to an integer? It does not, at least not in Pascal.

So the program above has a semantic error because the variable **y** is not declared and we don't know its type. In order for us to be able to catch errors like that, we need to learn how to check that variables are declared before they are used. So let's learn how to do it.

Let's take a closer look at the following syntactically and semantically correct sample program:

```
program Main;
   var x, y : integer;

begin
    x := x + y;
end.
```

- It has two variable declarations: **x** and **y**
- It also has three variable references (**x**, another **x**, and **y**) in the assignment statement **x := x + y**;

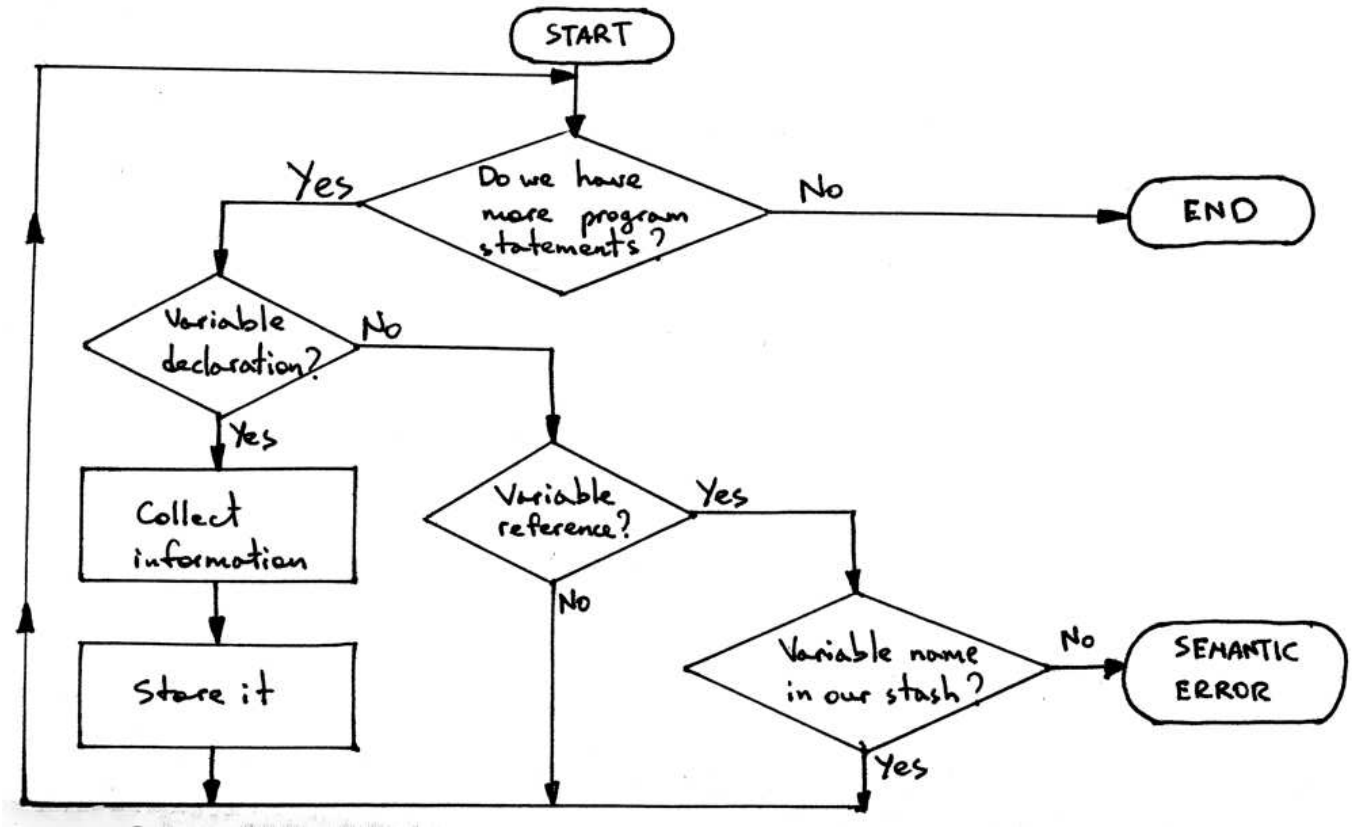


The program is grammatically correct, all the variables are declared, and we can see that adding two integers and assigning the result to an integer makes perfect sense. That's great, but how do we programmatically check that the variables (variable references) **x** and **y** in the assignment statement **x := x +y;** have been declared?

We can do this in several steps by implementing the following algorithm:

1. Go over all variable declarations
2. For every variable declaration you encounter, collect all necessary information about the declared variable
3. Store the collected information in some stash for future reference by using the variable's name as a key
4. When you see a variable reference, such as in the assignment statement **x := x + y**, search the stash by the variable's name to see if the stash has any information about the variable. If it does, the variable has been declared. If it doesn't, the variable hasn't been declared yet, which is a semantic error.

This is what a flowchart of our algorithm could look like:



Before we can implement the algorithm, we need to answer several questions:

- A. What information about variables do we need to collect?
- B. Where and how should we store the collected information?
- C. How do we implement the "go over all variable declarations" step?

Our plan of attack will be the following:

1. Figure out answers to the questions A, B, and C above.
2. Use the answers to A, B, and C to implement the steps in the algorithm for our first static semantic check: a check that variables are declared before they are used.

Okay, let's get started.

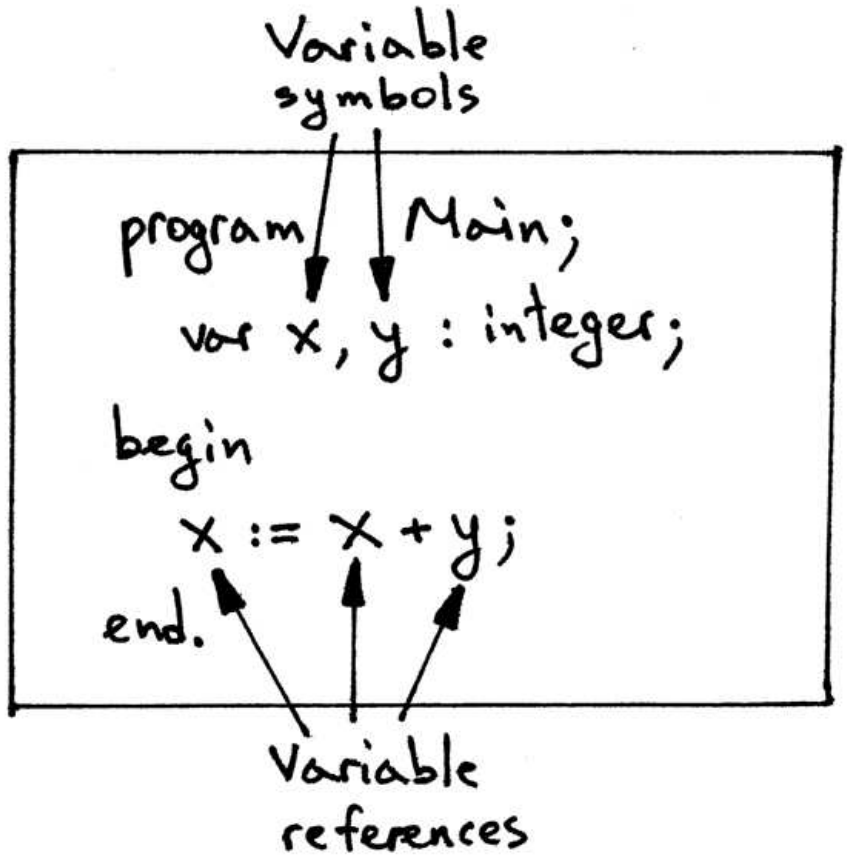
Let's find an answer to the question "What information about variables do we need to collect?"

So, what necessary information do we need to collect about a variable? Here are the important parts:

- *Name* (we need to know the name of a declared variable because later we will be looking up variables by their names)
- *Category* (we need to know what kind of an identifier it is: *variable*, *type*, *procedure*, and so on)
- *Type* (we'll need this information for type checking)

Symbols will hold that information (name, category, and type) about our variables. What's a *symbol*? A **symbol** is an identifier of some program entity like a variable, subroutine, or built-in type.

In the following sample program we have two variable declarations that we will use to create two variable symbols: **x**, and **y**.



In the code, we'll represent symbols with a class called *Symbol* that has fields *name* and *type* :

```
class Symbol(object):
    def __init__(self, name, type=None):
        self.name = name
        self.type = type
```

As you can see, the class takes the *name* parameter and an optional *type* parameter (not all symbols have type information associated with them, as we'll see shortly).

What about the *category*? We will encode *category* into the class name. Alternatively, we could store the category of a symbol in the dedicated *category* field of the *Symbol* class as in:

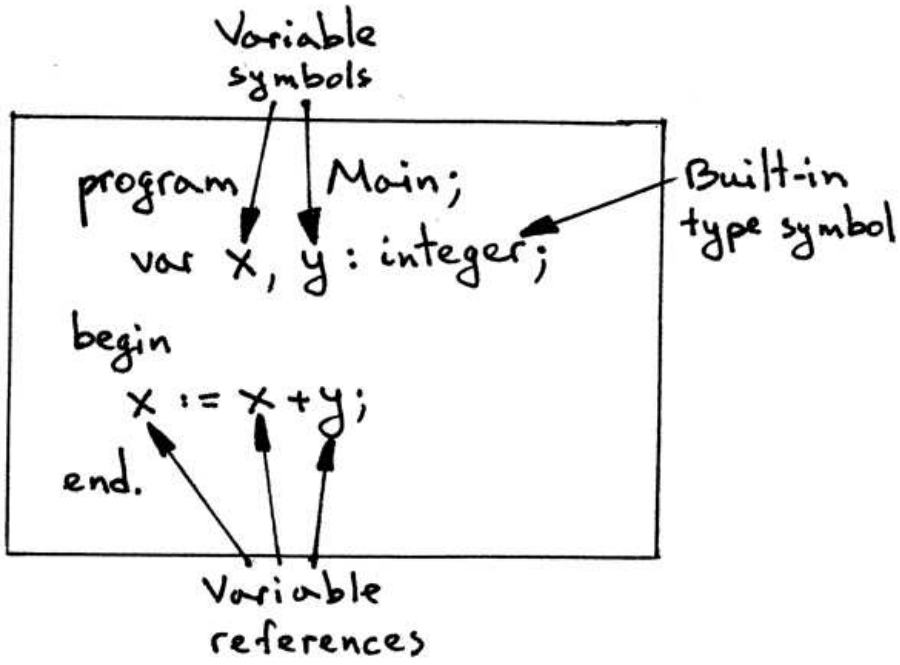
```
class Symbol(object):
    def __init__(self, name, type=None):
        self.name = name
        self.type = type
        self.category = category
```

However, it's more explicit to create a hierarchy of classes where the name of the class indicates its category.

Up until now I've sort of skirted around one topic, that of built-in types. If you look at our sample program again:

```
program Main;
   var x, y : integer;

begin
    x := x + y;
end.
```

You can see that variables **x** and **y** are declared as *integers*. What is the *integer* type? The integer type is another kind of symbol, a *built-in type symbol*. It's called built-in because it doesn't have to be declared explicitly in a Pascal program. It's our interpreter's responsibility to declare that type symbol and make it available to programmers:



We are going to make a separate class for built-in types called *BuiltinTypeSymbol*. Here is the class definition for our built-in types:

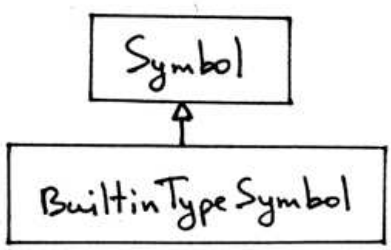
```
class BuiltinTypeSymbol(Symbol):
    def __init__(self, name):
        super(BuiltinTypeSymbol, self).__init__(name)

    def __str__(self):
        return self.name

    def __repr__(self):
        return "<{class_name}(name='{name}')>".format(
            class_name=self.__class__.__name__,
            name=self.name,
        )
```

The class *BuiltinTypeSymbol* inherits from the *Symbol* class, and its constructor requires only the *name* of the type, like *integer* or *real*. The 'builtin type' category is encoded in the class name, as we discussed earlier, and the *type* parameter from the base class is automatically set to *None* when we create a new instance of the *BuiltinTypeSymbol* class.

> ASIDE
>
> The double underscore or *dunder* (as in "**D**ouble **UNDER**score") methods *__str__* and *__repr__* are special Python methods. We've defined them to have a nice formatted message when we print a symbol object to standard output.

By the way, built-in types are the reason why the type parameter in the Scope class constructor is an optional parameter.

Here is our symbol class hierarchy so far:



Let's play with the builtin types in a Python shell. Download the interpreter file (https://github.com/rspivak/lsbasi/blob/master/part13/spi.py) and save it as spi.py; launch a python shell from the same directory where you saved the spi.py file, and play with the class we've just defined interactively:

```
$ python
>>> from spi import BuiltinTypeSymbol
>>> int_type = BuiltinTypeSymbol('integer')
>>> int_type
<BuiltinTypeSymbol(name='integer')>
>>>
>>> real_type = BuiltinTypeSymbol('real')
>>> real_type
<BuiltinTypeSymbol(name='real')>
```

That's all there is to built-in type symbols for now. Now back to our variable symbols.

How can we represent them in code? Let's create a *VarSymbol* class:

```
class VarSymbol(Symbol):
    def __init__(self, name, type):
        super(VarSymbol, self).__init__(name, type)

    def __str__(self):
        return "<{class_name}(name='{name}', type='{type}')>".format(
            class_name=self.__class__.__name__,
            name=self.name,
            type=self.type,
        )

    __repr__ = __str__
```

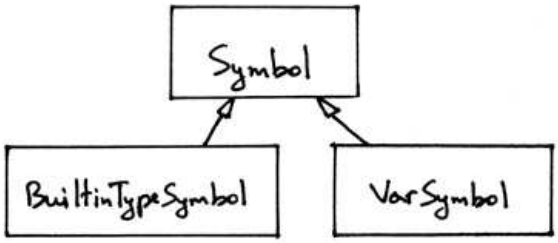
In this class, we made both the *name* and the *type* parameters required and the class name *VarSymbol* clearly indicates that an instance of the class will identify a variable symbol (the category is *variable*). The *type* parameter is an instance of the *BuiltinTypeSymbol* class.

Let's go back to the interactive Python shell to see how we can manually construct instances of our variable symbols now that we know how to construct *BuiltinTypeSymbol* class instances:

```
$ python
>>> from spi import BuiltinTypeSymbol, VarSymbol
>>> int_type = BuiltinTypeSymbol('integer')
>>> real_type = BuiltinTypeSymbol('real')
>>>
>>> var_x_symbol = VarSymbol('x', int_type)
>>> var_x_symbol
<VarSymbol(name='x', type='integer')>
>>>
>>> var_y_symbol = VarSymbol('y', real_type)
>>> var_y_symbol
<VarSymbol(name='y', type='real')>
>>>
```

As you can see, we first create an instance of the built-in type symbol and then pass it as a second parameter to *VarSymbol*'s constructor: variable symbols must have both a name and type associated with them as you've seen in various variable declarations like **var x : integer;**

And here is the complete hierarchy of symbols we've defined so far, in visual form:



Okay, now onto answering the question "Where and how should we store the collected information?"

Now that we have all the symbols representing all our variable declarations, where should we store those symbols so that we can search for them later when we encounter variable references (names)?

The answer is, as you probably already know, in *the symbol table*.

What is a *symbol table*? A **symbol table** is an abstract data type for tracking various symbols in source code. Think of it as a dictionary where the key is the symbol's name and the value is an instance of the symbol class (or one of its subclasses). To represent the symbol table in code we'll use a dedicated class for it aptly named *SymbolTable*. :) To store symbols in the symbol table we'll add the *insert* method to our symbol table class. The method *insert* will take a symbol as a parameter and store it internally in the *_symbols* ordered dictionary using the symbol's name as a key and the symbol instance as a value:

```
class SymbolTable(object):
    def __init__(self):
        self._symbols = OrderedDict()

    def __str__(self):
        symtab_header = 'Symbol table contents'
        lines = ['\n', symtab_header, '_' * len(symtab_header)]
        lines.extend(
            ('%7s: %r' % (key, value))
            for key, value in self._symbols.items()
        )
        lines.append('\n')
        s = '\n'.join(lines)
        return s

    __repr__ = __str__

    def insert(self, symbol):
        print('Insert: %s' % symbol.name)
        self._symbols[symbol.name] = symbol
```

Let's manually populate our symbol table for the following sample program. Because we don't know how to search our symbol table yet, our program won't contain any variable references, only variable declarations:

```
program SymTab1;
   var x, y : integer;

begin

end.
```

Download symtab01.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab01.py), which contains our new *SymbolTable* class and run it on the command line. This is what the output looks like for our program above:

```
$ python symtab01.py
Insert: INTEGER
Insert: x
Insert: y


Symbol table contents
_____________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>
```

And now let's build and populate the symbol table manually in a Python shell:

```
$ python
>>> from symtab01 import SymbolTable, BuiltinTypeSymbol, VarSymbol
>>> symtab = SymbolTable()
>>> int_type = BuiltinTypeSymbol('INTEGER')
>>> # now let's store the built-in type symbol in the symbol table
...
>>> symtab.insert(int_type)
Insert: INTEGER
>>>
>>> symtab


Symbol table contents
_________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>


>>> var_x_symbol = VarSymbol('x', int_type)
>>> symtab.insert(var_x_symbol)
Insert: x
>>> symtab


Symbol table contents
_________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
      x: <VarSymbol(name='x', type='INTEGER')>


>>> var_y_symbol = VarSymbol('y', int_type)
>>> symtab.insert(var_y_symbol)
Insert: y
>>> symtab


Symbol table contents
_________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


>>>
```

At this point we have answers to two questions that we asked earlier:

- A. What information about variables do we need to collect?

  Name, category, and type. And we use symbols to hold that information.

- B. Where and how should we store the collected information?

  We store collected symbols in the symbol table by using its insert method.

Now let's find the answer to our third question: ***'How do we implement the "go over all variable declarations" step?'***

This is a really easy one. Because we already have an AST built by our parser, we just need to create a new AST visitor class that will be responsible for walking over the tree and doing different actions when visiting *VarDecl* AST nodes!

Now we have answers to all three questions:

- A. What information about variables do we need to collect?

  Name, category, and type. And we use symbols to hold that information.

- B. Where and how should we store the collected information?

  We store collected symbols in the symbol table by using its *insert* method.

- C. How do we implement the "go over all variable declarations" step?

  We will create a new AST visitor that will do some actions on visiting *VarDecl* AST nodes.

Let's create a new tree visitor class and give it the name *SemanticAnalyzer*. Take a look the following sample program, for example:

```
program SymTab2;
   var x, y : integer;

begin

end.
```

To be able to analyze the program above, we don't need to implement all *visit_xxx* methods, just a subset of them. Below is the skeleton for the *SemanticAnalyzer* class with enough *visit_xxx* methods to be able to successfully walk the AST of the sample program above:

```
class SemanticAnalyzer(NodeVisitor):
    def __init__(self):
        self.symtab = SymbolTable()

    def visit_Block(self, node):
        for declaration in node.declarations:
            self.visit(declaration)
        self.visit(node.compound_statement)

    def visit_Program(self, node):
        self.visit(node.block)

    def visit_Compound(self, node):
        for child in node.children:
            self.visit(child)

    def visit_NoOp(self, node):
        pass

    def visit_VarDecl(self, node):
        #  Actions go here
        pass
```

Now, we have all the pieces to implement the first three steps of our algorithm for our first static semantic check, the check that verifies that variables are declared before they are used.

Here are the steps of the algorithm again:

1. Go over all variable declarations
2. For every variable declaration you encounter, collect all necessary information about the declared variable
3. Store the collected information in some stash for future references by using the variable's name as a key
4. When you see a variable reference such as in the assignment statement **x := x + y**, search the stash by the variable's name to see if the stash has any information about the variable. If it does, the variable has been declared. If it doesn't, the variable hasn't been declared yet, which is a semantic error.

Let's implement those steps. Actually, the only thing that we need to do is fill in the *visit_VarDecl* method of the *SemanticAnalyzer* class. Here it is, filled in:

```
def visit_VarDecl(self, node):
    # For now, manually create a symbol for the INTEGER built-in type
    # and insert the type symbol in the symbol table.
    type_symbol = BuiltinTypeSymbol('INTEGER')
    self.symtab.insert(type_symbol)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)
    self.symtab.insert(var_symbol)
```

If you look at the contents of the method, you can see that it actually incorporates all three steps:

1. The method will be called for *every* variable declaration once we've invoked the *visit* method of the *SemanticAnalyzer* instance. That covers Step 1 of the algorithm: *"Go over all variable declarations"*
2. For every variable declaration, the method *visit_VarDecl* will collect the necessary information and create a variable symbol instance. That covers Step 2 of the algorithm: *"For every variable declaration you encounter, collect all necessary information about the declared variable"*
3. The method *visit_VarDecl* will store the collected information about the variable declaration in the symbol table using the symbol table's *insert* method. This covers Step 3 of the algorithm: *"Store the collected information in some stash for future references by using the variable's name as a key"*

To see all of those steps in action, download file symtab02.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab02.py) and study its source code first. Then run it on the command line and inspect the output:

```
$ python symtab02.py
Insert: INTEGER
Insert: x
Insert: INTEGER
Insert: y


Symbol table contents
________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>
```

You might have noticed that there are two lines that say *Insert: INTEGER*. We will fix this situation in the following section where we'll discuss the implementation of the final step (Step 4) of the semantic check algorithm.

Okay, let's implement Step 4 of our algorithm. Here is an updated version of Step 4 to reflect the introduction of symbols and the symbol table: *When you see a variable reference (name) such as in the assignment statement **x := x + y**, search the symbol table by the variable's name to see if the table has a variable symbol associated with the name. If it does, the variable has been declared. If it doesn't, the variable hasn't been declared yet, which is a semantic error.*

To implement Step 4, we need to make some changes to the symbol table and semantic analyzer:

1. We need to add a method to our symbol table that will be able to look up a symbol by name.
2. We need to update our semantic analyzer to look up a name in the symbol table every time it encounters a variable reference.

First, let's update our *SymbolTable* class by adding the *lookup* method that will be responsible for searching for a symbol by name. In other words, the *lookup* method will be responsible for resolving a variable name (a variable reference) to its declaration. The process of mapping a variable reference to its declaration is called **name resolution**. And here is our *lookup* method that does just that, *name resolution*:

```
def lookup(self, name):
    print('Lookup: %s' % name)
    symbol = self._symbols.get(name)
    # 'symbol' is either an instance of the Symbol class or None
    return symbol
```

The method takes a symbol name as a parameter and returns a symbol if it finds it or *None* if it doesn't. As simple as that.

While we're at it, let's also update our *SymbolTable* class to initialize built-in types. We'll do that by adding a method *_init_builtins* and calling it in the *SymbolTable*'s constructor. The *_init_builtins* method will insert a type symbol for *integer* and a type symbol for *real* into the symbol table.

Here is the full code for our updated *SymbolTable* class:

```
class SymbolTable(object):
    def __init__(self):
        self._symbols = OrderedDict()
        self._init_builtins()

    def _init_builtins(self):
        self.insert(BuiltinTypeSymbol('INTEGER'))
        self.insert(BuiltinTypeSymbol('REAL'))

    def __str__(self):
        symtab_header = 'Symbol table contents'
        lines = ['\n', symtab_header, '_' * len(symtab_header)]
        lines.extend(
            ('%7s: %r' % (key, value))
            for key, value in self._symbols.items()
        )
        lines.append('\n')
        s = '\n'.join(lines)
        return s

    __repr__ = __str__

    def insert(self, symbol):
        print('Insert: %s' % symbol.name)
        self._symbols[symbol.name] = symbol

    def lookup(self, name):
        print('Lookup: %s' % name)
        symbol = self._symbols.get(name)
        # 'symbol' is either an instance of the Symbol class or None
        return symbol
```

Now that we have built-in type symbols and the *lookup* method to search our symbol table when we encounter variable names (and other names like type names), let's update the *SemanticAnalyzer*'s *visit_VarDecl* method and replace the two lines where we were manually creating the INTEGER built-in type symbol and manually inserting it into the symbol table with code to look up the INTEGER type symbol.

The change will also fix the issue with that double output of the *Insert: INTEGER* line we've seen before.

Here is the *visit_VarDecl* method before the change:

```
def visit_VarDecl(self, node):
    # For now, manually create a symbol for the INTEGER built-in type
    # and insert the type symbol in the symbol table.
    type_symbol = BuiltinTypeSymbol('INTEGER')
    self.symtab.insert(type_symbol)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)
    self.symtab.insert(var_symbol)
```

and after the change:

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.symtab.lookup(type_name)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)
    self.symtab.insert(var_symbol)
```

Let's apply the changes to the familiar Pascal program that has only variable declarations:

```
program SymTab3;
   var x, y : integer;

begin

end.
```

Download the symtab03.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab03.py) file that has all the changes we've just discussed, run it on the command line, and see that there is no longer a duplicate *Insert: INTEGER* line in the program output any more:

```
$ python symtab03.py
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: x
Lookup: INTEGER
Insert: y


Symbol table contents
________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>
```

You can also see in the output above that our semantic analyzer looks up the *INTEGER* built-in type twice: first for the declaration of the variable **x**, and the second time for the declaration of the variable **y**.

Now let's switch our attention to variable references (names) and how we can resolve a variable name, let's say in an arithmetic expression, to its variable declaration (variable symbol). Let's take a look at the following sample program, for example, that has an assignment statement **x := x + y;** with three variable references: **x**, another **x**, and **y**:

```
program SymTab4;
    var x, y : integer;

begin
    x := x + y;
end.
```

We already have the *lookup* method in our symbol table implementation. What we need to do now is extend our semantic analyzer so that every time it encounters a variable reference it would search the symbol table by the variable reference name using the symbol table's *lookup* name. What method of the *SemanticAnalyzer* gets called every time a variable reference is encountered when the analyzer walks the AST? It's the method *visit_Var*. Let's add it to our class. It's very simple: all it does is look up the variable symbol by name:

```
def visit_Var(self, node):
    var_name = node.value
    var_symbol = self.symtab.lookup(var_name)
```

Because our sample program *SymTab4* has an assignment statement with arithmetic addition in its right hand side, we need to add two more methods to our *SemanticAnalyzer* so that it could actually walk the AST of the *SymTab4* program and call the *visit_Var* method for all *Var* nodes. The methods we need to add are *visit_Assign* and *visit_BinOp*. They are nothing new: you've seen these methods before. Here they are:

```
def visit_Assign(self, node):
    # right-hand side
    self.visit(node.right)
    # left-hand side
    self.visit(node.left)

def visit_BinOp(self, node):
    self.visit(node.left)
    self.visit(node.right)
```

You can find the full source code with the changes we've just discussed in the file symtab04.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab04.py). Download the file, run it on the command line, and inspect the output produced for our sample program *SymTab4* with an assignment statement.

Here is the output on my laptop:

```
$ python symtab04.py
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: x
Lookup: INTEGER
Insert: y
Lookup: x
Lookup: y
Lookup: x


Symbol table contents
________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>
```

Spend some time analyzing the output and making sure you understand how and why the output is generated in that order.

At this point, we have implemented all of the steps of our algorithm for a static semantic check that verifies that all variables in the program are declared before they are used!

Semantic errors

So far we've looked at the programs that had their variables declared, but what if our program has a variable reference that doesn't resolve to any declaration; that is, it's not declared? That's a semantic error and we need to extend our semantic analyzer to signal that error.

Take a look at the following semantically incorrect program, where the variable **y** is not declared but used in the assignment statement:

```
program SymTab5;
    var x : integer;

begin
    x := y;
end.
```

To signal the error, we need to modify our *SemanticAnalyzer*'s *visit_Var* method to throw an exception if the *lookup* method cannot resolve a name to a symbol and returns *None*. Here is the updated code for *visit_Var*:

```
def visit_Var(self, node):
    var_name = node.value
    var_symbol = self.symtab.lookup(var_name)
    if var_symbol is None:
        raise Exception(
            "Error: Symbol(identifier) not found '%s'" % var_name
        )
```

Download symtab05.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab05.py), run it on the command line, and see what happens:

```
$ python symtab05.py
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: x
Lookup: y
Error: Symbol(identifier) not found 'y'


Symbol table contents
________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='INTEGER')>
```

You can see the error message ***Error: Symbol(identifier) not found 'y'*** and the contents of the symbol table.

Congratulations on finishing the current version of our semantic analyzer that can statically check if variables in a program are declared before they are used, and if they are not, throws an exception indicating a semantic error!

Let's pause for a second and celebrate this important milestone. Okay, the second is over and we need to move on to another static semantic check. For fun and profit let's extend our semantic analyzer to check for duplicate identifiers in declarations.

Let's take a look at the following program, SymTab6:

```
program SymTab6;
   var x, y : integer;
   var y : real;
begin
   x := x + y;
end.
```

Variable **y** has been declared twice: the first time as *integer* and the second time as *real*.

To catch that semantic error we need to modify our *visit_VarDecl* method to check whether the symbol table already has a symbol with the same name before inserting a new symbol. Here is our new version of the method:

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.symtab.lookup(type_name)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)

    # Signal an error if the table alrady has a symbol
    # with the same name
    if self.symtab.lookup(var_name) is not None:
        raise Exception(
            "Error: Duplicate identifier '%s' found" % var_name
        )

    self.symtab.insert(var_symbol)
```

File symtab06.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab06.py) has all the changes. Download it and run it on the command line:

```
$ python symtab06.py
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Lookup: x
Insert: x
Lookup: INTEGER
Lookup: y
Insert: y
Lookup: REAL
Lookup: y
Error: Duplicate identifier 'y' found


Symbol table contents
________________________
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>
```

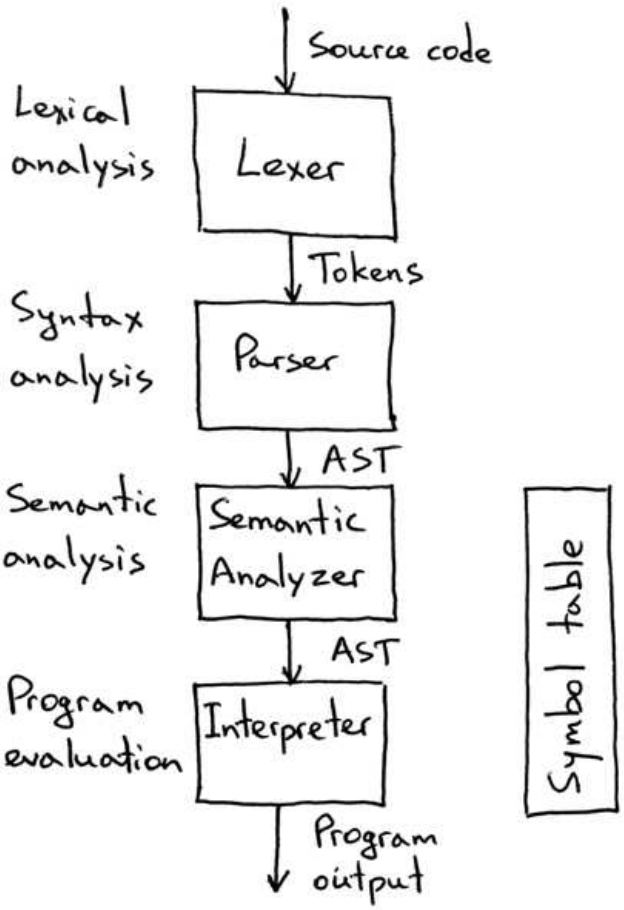
Study the output and the contents of the symbol table. Make sure you understand what's going on.

Summary

Let's quickly recap what we learned today:

- We learned more about symbols, symbol tables, and semantic analysis in general
- We learned about name resolution and how the semantic analyzer resolves names to their declarations
- We learned how to code a semantic analyzer that walks an AST, builds the symbol table, and does basic semantic checks

And, as a reminder, the structure of our interpreter now looks like this:



We're done with semantic checks for today and we're finally ready to tackle the topic of scopes, how they relate to symbol tables, and the topic of semantic checks in the presence of nested scopes. Those will be central topics of the next article. Stay tuned and see you soon!

All articles in this series:

- Let's Build A Simple Interpreter. Part 1. (/lsbasi-part1/)
- Let's Build A Simple Interpreter. Part 2. (/lsbasi-part2/)
- Let's Build A Simple Interpreter. Part 3. (/lsbasi-part3/)
- Let's Build A Simple Interpreter. Part 4. (/lsbasi-part4/)
- Let's Build A Simple Interpreter. Part 5. (/lsbasi-part5/)
- Let's Build A Simple Interpreter. Part 6. (/lsbasi-part6/)
- Let's Build A Simple Interpreter. Part 7. (/lsbasi-part7/)
- Let's Build A Simple Interpreter. Part 8. (/lsbasi-part8/)
- Let's Build A Simple Interpreter. Part 9. (/lsbasi-part9/)
- Let's Build A Simple Interpreter. Part 10. (/lsbasi-part10/)
- Let's Build A Simple Interpreter. Part 11. (/lsbasi-part11/)
- Let's Build A Simple Interpreter. Part 12. (/lsbasi-part12/)
- Let's Build A Simple Interpreter. Part 13. (/lsbasi-part13/)
- Let's Build A Simple Interpreter. Part 14. (/lsbasi-part14/)

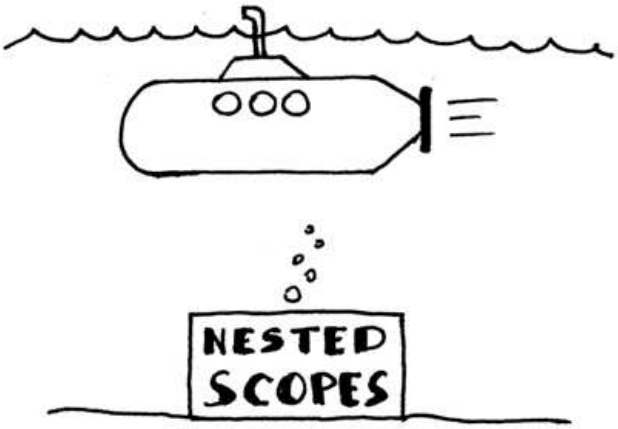
Comments

Let’s Build A Simple Interpreter. Part 14: Nested Scopes and a Source-to-Source Compiler. (https://ruslanspivak.com/lsbasi-part14/)

Date Mon, May 08, 2017

> *Only dead fish go with the flow.*

As I promised in the last article (/lsbasi-part13), today we’re finally going to do a deep dive into the topic of scopes.



This is what we’re going to learn today:

- We’re going to learn about *scopes*, why they are useful, and how to implement them in code with symbol tables.
- We’re going to learn about *nested scopes* and how *chained scoped symbol tables* are used to implement nested scopes.
- We’re going to learn how to parse procedure declarations with formal parameters and how to represent a procedure symbol in code.
- We’re going to learn how to extend our *semantic analyzer* to do semantic checks in the presence of nested scopes.
- We’re going to learn more about *name resolution* and how the semantic analyzer resolves names to their declarations when a program has nested scopes.
- We’re going to learn how to build a *scope tree*.
- We’re also going to learn how to write our very own ***source-to-source compiler*** today! We will see later in the article how relevant it is to our discussion of scopes.

Let’s get started! Or should I say, let’s dive in!

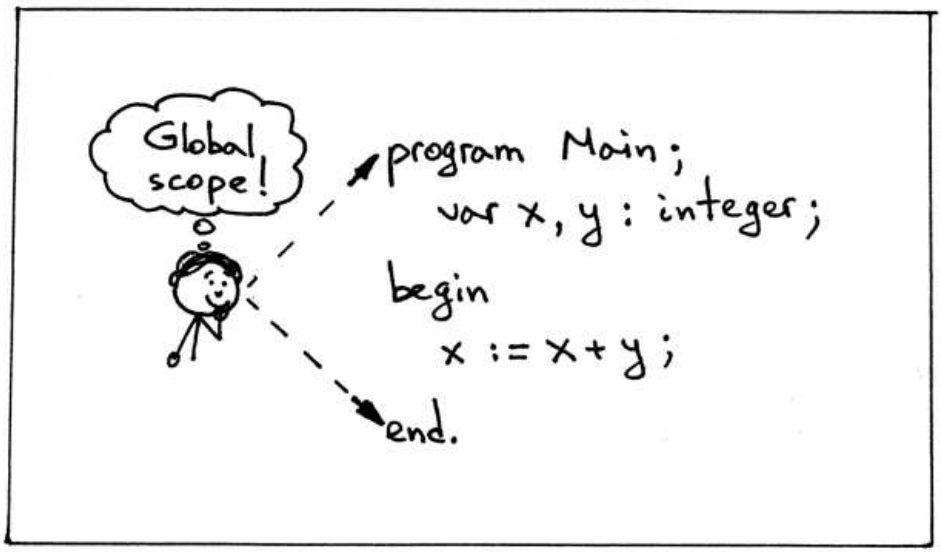
Table of Contents

Scopes and scoped symbol tables

What is a *scope*? A ***scope*** is a textual region of a program where a name can be used. Let's take a look at the following sample program, for example:

```
program Main;
   var x, y: integer;
begin
   x := x + y;
end.
```

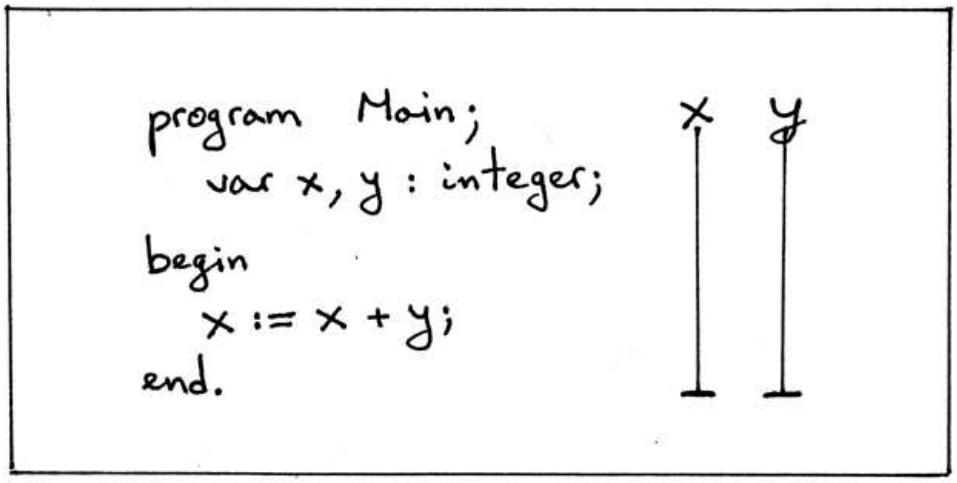
In Pascal, the *PROGRAM* keyword (case insensitive, by the way) introduces a new scope which is commonly called a *global scope*, so the program above has one *global scope* and the declared variables **x** and **y** are visible and accessible in the whole program. In the case above, the textual region starts with the keyword *program* and ends with the keyword *end* and a dot. In that textual region both names **x** and **y** can be used, so the scope of those variables (variable declarations) is the whole program:



When you look at the source code above and specifically at the expression **x := x + y**, you intuitively know that it should compile (or get interpreted) without a problem, because the scope of the variables **x** and **y** in the expression is the *global scope* and the variable references **x** and **y** in the expression **x**

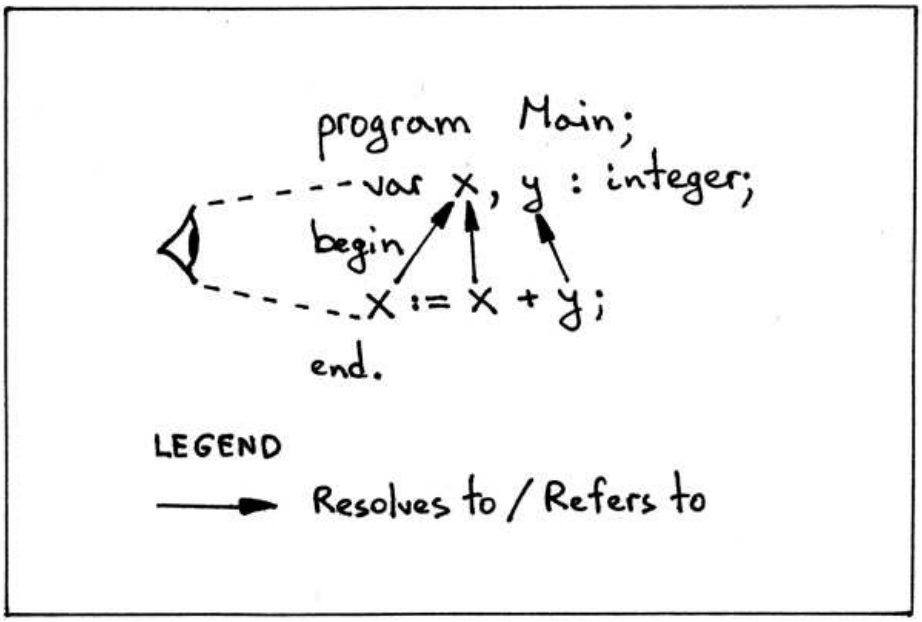
:= x + y resolve to the declared integer variables **x** and **y**. If you've programmed before in any mainstream programming language, there shouldn't be any surprises here.

When we talk about the scope of a variable, we actually talk about the scope of its declaration:



In the picture above, the vertical lines show the scope of the declared variables, the textual region where the declared names **x** and **y** can be used, that is, the text area where they are visible. And as you can see, the scope of **x** and **y** is the whole program, as shown by the vertical lines.

Pascal programs are said to be ***lexically scoped*** (or ***statically scoped***) because you can look at the source code, and without even executing the program, determine purely based on the textual rules which names (references) resolve or refer to which declarations. In Pascal, for example, lexical keywords like *program* and *end* demarcate the textual boundaries of a scope:



Why are scopes useful?

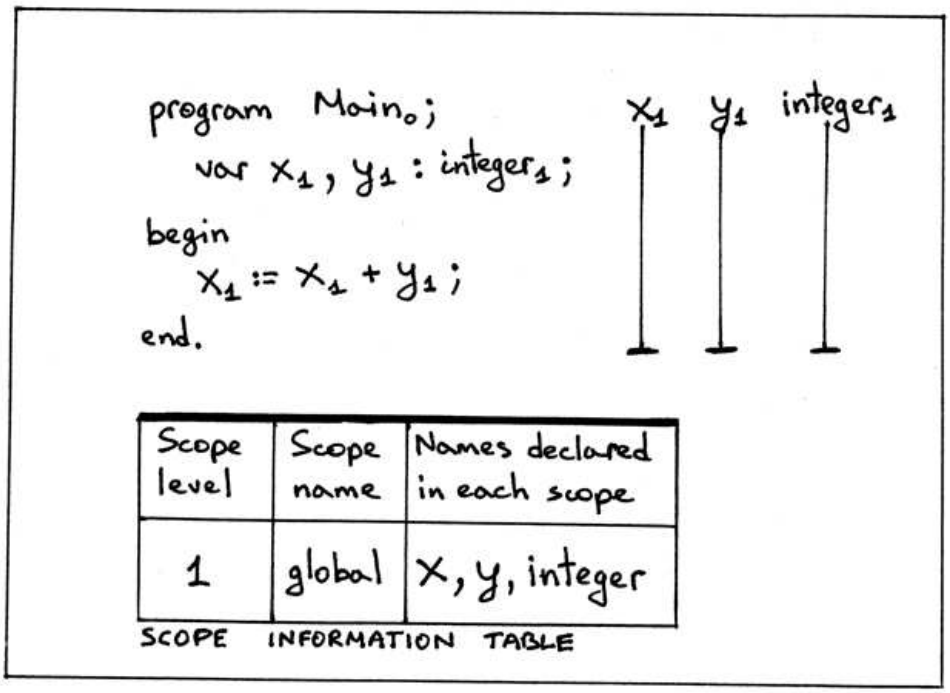
- Every scope creates an isolated name space, which means that variables declared in a scope cannot be accessed from outside of it.
- You can re-use the same name in different scopes and know exactly, just by looking at the program source code, what declaration the name refers to at every point in the program.
- In a nested scope you can re-declare a variable with the same name as in the outer scope, thus effectively hiding the outer declaration, which gives you control over access to different variables from the outer scope.

In addition to the *global scope*, Pascal supports nested procedures, and every procedure declaration introduces a new scope, which means that Pascal supports nested scopes.

When we talk about nested scopes, it's convenient to talk about scope levels to show their nesting relationships. It's also convenient to refer to scopes by name. We'll use both scope levels and scope names when we start our discussion of nested scopes.

Let's take a look at the following sample program and subscript every name in the program to make it clear:

1. At what level each variable (symbol) is declared
2. To which declaration and at what level a variable name refers to:



From the picture above we can see several things:

- We have a single scope, the *global scope*, introduced by the PROGRAM keyword
- *Global scope* is at level 1
- Variables (symbols) **x** and **y** are declared at level 1 (the *global scope*).
- *integer* built-in type is also declared at level 1
- The program name **Main** has a subscript 0. Why is the program's name at level zero, you might wonder? This is to make it clear that the program's name is not in the *global scope* and it's in some other outer scope, that has level zero.
- The scope of the variables **x** and **y** is the whole program, as shown by the vertical lines
- The *scope information table* shows for every level in the program the corresponding scope level, scope name, and names declared in the scope. The purpose of the table is to summarize and visually show different information about scopes in a program.

How do we implement the concept of a scope in code? To represent a scope in code, we'll need a *scoped symbol table*. We already know about symbol tables, but what is a *scoped symbol table*? A ***scoped symbol table*** is basically a symbol table with a few modifications, as you'll see shortly.

From now on, we'll use the word *scope* both to mean the concept of a scope as well as to refer to the scoped symbol table, which is an implementation of the scope in code.

Even though in our code a scope is represented by an instance of the *ScopedSymbolTable* class, we'll use the variable named *scope* throughout the code for convenience. So when you see a variable *scope* in the code of our interpreter, you should know that it actually refers to a *scoped symbol table*.

Okay, let's enhance our *SymbolTable* class by renaming it to *ScopedSymbolTable* class, adding two new fields *scope_level* and *scope_name*, and updating the scoped symbol table's constructor. And at the same time, let's update the *__str__* method to print additional information, namely the *scope_level* and *scope_name*. Here is a new version of the symbol table, the *ScopedSymbolTable*:

```
class ScopedSymbolTable(object):
    def __init__(self, scope_name, scope_level):
        self._symbols = OrderedDict()
        self.scope_name = scope_name
        self.scope_level = scope_level
        self._init_builtins()

    def _init_builtins(self):
        self.insert(BuiltinTypeSymbol('INTEGER'))
        self.insert(BuiltinTypeSymbol('REAL'))

    def __str__(self):
        h1 = 'SCOPE (SCOPED SYMBOL TABLE)'
        lines = ['\n', h1, '=' * len(h1)]
        for header_name, header_value in (
            ('Scope name', self.scope_name),
            ('Scope level', self.scope_level),
        ):
            lines.append('%-15s: %s' % (header_name, header_value))
        h2 = 'Scope (Scoped symbol table) contents'
        lines.extend([h2, '-' * len(h2)])
        lines.extend(
            ('%7s: %r' % (key, value))
            for key, value in self._symbols.items()
        )
        lines.append('\n')
        s = '\n'.join(lines)
        return s

    __repr__ = __str__

    def insert(self, symbol):
        print('Insert: %s' % symbol.name)
        self._symbols[symbol.name] = symbol

    def lookup(self, name):
        print('Lookup: %s' % name)
        symbol = self._symbols.get(name)
        # 'symbol' is either an instance of the Symbol class or None
        return symbol
```

Let's also update the semantic analyzer's code to use the variable *scope* instead of *symtab*, and remove the semantic check that was checking source programs for duplicate identifiers from the *visit_VarDecl* method to reduce the noise in the program output.

Here is a piece of code that shows how our semantic analyzer instantiates the *ScopedSymbolTable* class:

```
class SemanticAnalyzer(NodeVisitor):
    def __init__(self):
        self.scope = ScopedSymbolTable(scope_name='global', scope_level=1)

    ...
```

You can find all the changes in the file scope01.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope01.py). Download the file, run it on the command line, and inspect the output. Here is what I got:

```
$ python scope01.py
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: x
Lookup: INTEGER
Insert: y
Lookup: x
Lookup: y
Lookup: x


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>
```

Most of the output should look very familiar to you.

Now that you know about the concept of scope and how to implement the scope in code by using a scoped symbol table, it's time we talked about nested scopes and more dramatic modifications to the scoped symbol table than just adding two simple fields.

Procedure declarations with formal parameters

Let's take a look at a sample program in the file nestedscopes02.pas (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes02.pas) that contains a procedure declaration:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
   begin
      x := a + x + y;
   end;

begin { Main }

end.  { Main }
```

The first thing that we notice here is that we have a procedure with a parameter, and we haven't learned how to handle that yet. Let's fill that gap by making a quick detour and learning how to handle formal procedure parameters before continuing with scopes.*

> *ASIDE: *Formal parameters* are parameters that show up in the declaration of a procedure. *Arguments* (also called *actual parameters*) are different variables and expressions passed to a procedure in a particular procedure call.

Here is a list of changes we need to make to support procedure declarations with parameters:

1. Add the *Param* AST node

```
class Param(AST):
    def __init__(self, var_node, type_node):
        self.var_node = var_node
        self.type_node = type_node
```

2. Update the *ProcedureDecl* node's constructor to take an additional argument: *params*

```
class ProcedureDecl(AST):
    def __init__(self, proc_name, params, block_node):
        self.proc_name = proc_name
        self.params = params  # a list of Param nodes
        self.block_node = block_node
```

3. Update the *declarations* rule to reflect changes in the procedure declaration sub-rule

```
def declarations(self):
    """declarations : (VAR (variable_declaration SEMI)+)*
                    | (PROCEDURE ID (LPAREN formal_parameter_list RPAREN)? SEMI block SEMI)
                    | empty
    """
```

4. Add the *formal_parameter_list* rule and method

```
def formal_parameter_list(self):
    """ formal_parameter_list : formal_parameters
                              | formal_parameters SEMI formal_parameter_list
    """
```

5. Add the *formal_parameters* rule and method

```
def formal_parameters(self):
    """ formal_parameters : ID (COMMA ID)* COLON type_spec """
    param_nodes = []
```

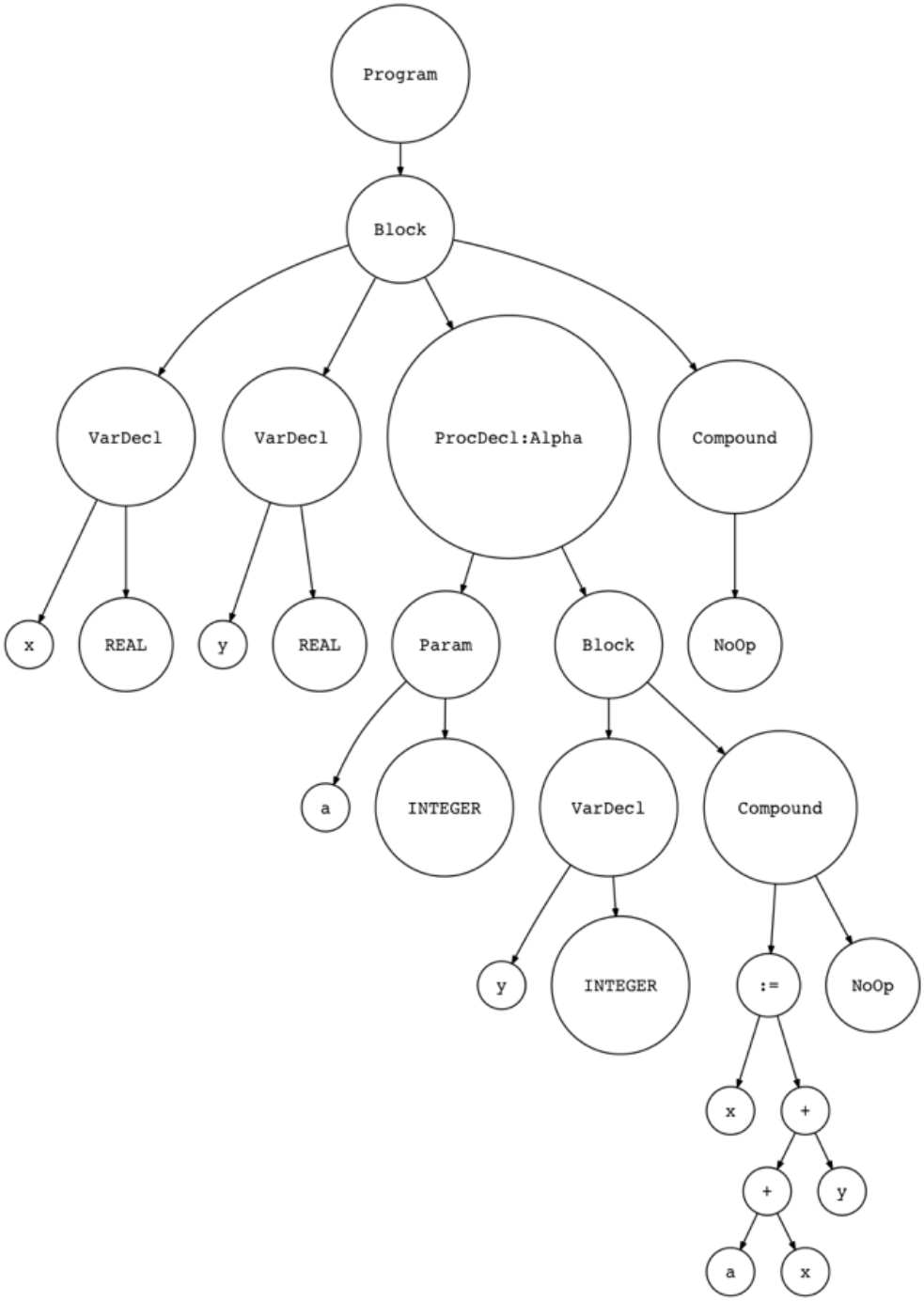
With the addition of the above methods and rules our parser will be able to parse procedure declarations like these (I'm not showing the body of declared procedures for brevity):

```
procedure Foo;

procedure Foo(a : INTEGER);

procedure Foo(a, b : INTEGER);

procedure Foo(a, b : INTEGER; c : REAL);
```

Let's generate an AST for our sample program. Download genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part14/genastdot.py) and run the following command on the command line:

```
$ python genastdot.py nestedscopes02.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```

Here is a picture of the generated AST:



You can see now that the *ProcedureDecl* node in the picture has the *Param* node as its child.

You can find the complete changes in the spi.py (https://github.com/rspivak/lsbasi/blob/master/part14/spi.py) file. Spend some time and study the changes. You've done similar changes before; they should be pretty easy to understand and you should be able to implement them by yourself.

Procedure symbols

While we're on the topic of procedure declarations, let's also talk about procedure symbols.

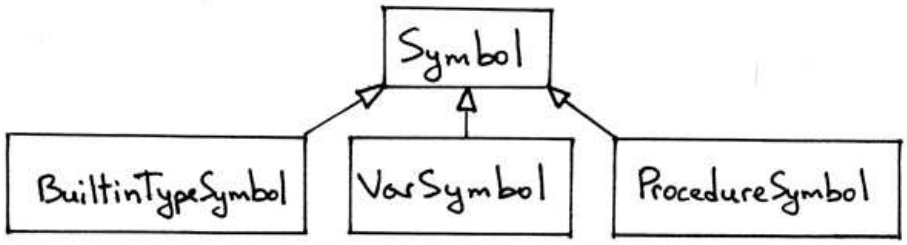
As with variable declarations, and built-in type declarations, there is a separate category of symbols for procedures. Let's create a separate symbol class for procedure symbols:

```
class ProcedureSymbol(Symbol):
    def __init__(self, name, params=None):
        super(ProcedureSymbol, self).__init__(name)
        # a list of formal parameters
        self.params = params if params is not None else []

    def __str__(self):
        return '<{class_name}(name={name}, parameters={params})>'.format(
            class_name=self.__class__.__name__,
            name=self.name,
            params=self.params,
        )

    __repr__ = __str__
```

Procedure symbols have a name (it's a procedure's name), their category is procedure (it's encoded in the class name), and the type is *None* because in Pascal procedures don't return anything.

Procedure symbols also carry additional information about procedure declarations, namely they contain information about the procedure's formal parameters as you can see in the code above.

With the addition of procedure symbols, our new symbol hierarchy looks like this:



Nested scopes

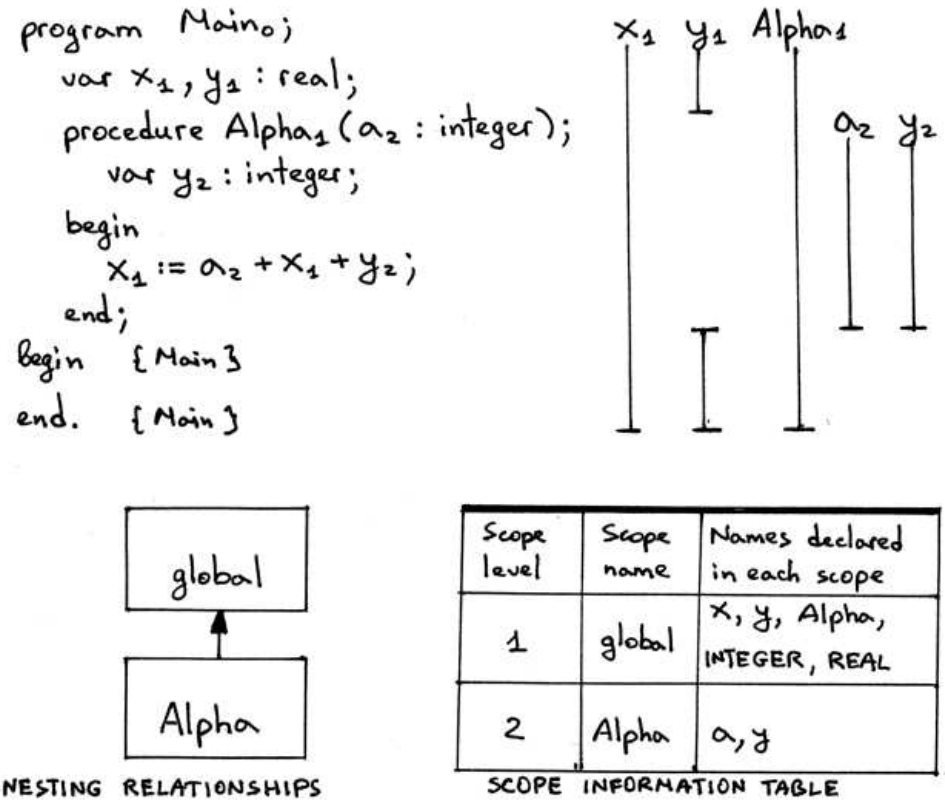
After that quick detour let's get back to our program and the discussion of nested scopes:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
   begin
      x := a + x + y;
   end;

begin { Main }

end.  { Main }
```

Things are actually getting more interesting here. By declaring a new procedure, we introduce a new scope, and this scope is nested within the *global scope* introduced by the *PROGRAM* statement, so this is a case where we have nested scopes in a Pascal program.

The scope of a procedure is the whole body of the procedure. The beginning of the procedure scope is marked by the *PROCEDURE* keyword and the end is marked by the *END* keyword and a semicolon.

Let's subscript names in the program and show some additional information:



Some observations from the picture above:

- This Pascal program has two scope levels: level 1 and level 2
- The *nesting relationships* diagram visually shows that the scope *Alpha* is nested within the *global scope*, hence there are two levels: the *global scope* at level 1, and the *Alpha* scope at level 2.
- The scope level of the procedure declaration *Alpha* is one less than the level of the variables declared inside the procedure *Alpha*. You can see that the scope level of the procedure declaration *Alpha* is 1 and the scope level of the variables **a** and **y** inside the procedure is 2.
- The variable declaration of **y** inside *Alpha* hides the declaration of **y** in the *global scope*. You can see the hole in the vertical bar for **y1** (by the way, 1 is a subscript, it's not part of the variable name, the variable name is just **y**) and you can see that the scope of the **y2** variable declaration is the *Alpha* procedure's whole body.
- The scope information table, as you are already aware, shows scope levels, scope names for those levels, and respective names declared in those scopes (at those levels).
- In the picture, you can also see that I omitted showing the scope of the *integer* and *real* types (except in the scope information table) because they are always declared at scope level 1, the *global scope*, so I won't be subscripting the *integer* and *real* types anymore to save visual space, but you will see the types again and again in the contents of the scoped symbol table representing the *global scope*.

The next step is to discuss implementation details.

First, let's focus on variable and procedure declarations. Then, we'll discuss variable references and how *name resolution* works in the presence of nested scopes.

For our discussion, we'll use a stripped down version of the program. The following version does not have variable references: it only has variable and procedure declarations:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
   begin

   end;

begin { Main }

end.  { Main }
```

You already know how to represent a scope in code with a scoped symbol table. Now we have two scopes: the *global scope* and the scope introduced by the procedure *Alpha*. Following our approach we should now have two scoped symbol tables: one for the *global scope* and one for the *Alpha* scope. How do we implement that in code? We'll extend the semantic analyzer to create a separate scoped symbol table for every scope instead of just for the *global scope*. The scope construction will happen, as usual, when walking the AST.

First, we need to decide where in the semantic analyzer we're going to create our scoped symbol tables. Recall that *PROGRAM* and *PROCEDURE* keywords introduce new scope. In AST, the corresponding nodes are *Program* and *ProcedureDecl*. So we're going to update our *visit_Program* method and add the *visit_ProcedureDecl* method to create scoped symbol tables. Let's start with the *visit_Program* method:

```
def visit_Program(self, node):
    print('ENTER scope: global')
    global_scope = ScopedSymbolTable(
        scope_name='global',
        scope_level=1,
    )
    self.current_scope = global_scope

    # visit subtree
    self.visit(node.block)

    print(global_scope)
    print('LEAVE scope: global')
```

The method has quite a few changes:

1. When visiting the node in AST, we first print what scope we're entering, in this case *global*.
2. We create a separate *scoped symbol table* to represent the *global scope*. When we construct an instance of *ScopedSymbolTable*, we explicitly pass the scope name and scope level arguments to the class constructor.
3. We assign the newly created scope to the instance variable *current_scope*. Other visitor methods that insert and look up symbols in scoped symbol tables will use the *current_scope*.
4. We visit a subtree (block). This is the old part.
5. Before leaving the *global scope* we print the contents of the *global scope* (scoped symbol table)
6. We also print the message that we're leaving the *global scope*

Now let's add the *visit_ProcedureDecl* method. Here is the complete source code for it:

```
def visit_ProcedureDecl(self, node):
    proc_name = node.proc_name
    proc_symbol = ProcedureSymbol(proc_name)
    self.current_scope.insert(proc_symbol)

    print('ENTER scope: %s' %  proc_name)
    # Scope for parameters and local variables
    procedure_scope = ScopedSymbolTable(
        scope_name=proc_name,
        scope_level=2,
    )
    self.current_scope = procedure_scope

    # Insert parameters into the procedure scope
    for param in node.params:
        param_type = self.current_scope.lookup(param.type_node.value)
        param_name = param.var_node.value
        var_symbol = VarSymbol(param_name, param_type)
        self.current_scope.insert(var_symbol)
        proc_symbol.params.append(var_symbol)

    self.visit(node.block_node)

    print(procedure_scope)
    print('LEAVE scope: %s' %  proc_name)
```

Let's go over the contents of the method:

1. The first thing that the method does is create a procedure symbol and insert it into the current scope, which is the *global scope* for our sample program.
2. Then the method prints the message about entering the procedure scope.
3. Then we create a new scope for the procedure's parameters and variable declarations.
4. We assign the procedure scope to the *self.current_scope* variable indicating that this is our current scope and all symbol operations (*insert* and *lookup*) will use the current scope.
5. Then we handle procedure formal parameters by inserting them into the current scope and adding them to the procedure symbol.
6. Then we visit the rest of the AST subtree - the body of the procedure.
7. And, finally, we print the message about leaving the scope before leaving the node and moving to another AST node, if any.

Now, what we need to do is update other semantic analyzer visitor methods to use *self.current_scope* when inserting and looking up symbols. Let's do that:

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.current_scope.lookup(type_name)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)

    self.current_scope.insert(var_symbol)

def visit_Var(self, node):
    var_name = node.value
    var_symbol = self.current_scope.lookup(var_name)
    if var_symbol is None:
        raise Exception(
            "Error: Symbol(identifier) not found '%s'" % var_name
        )
```

Both the *visit_VarDecl* and *visit_Var* will now use the *current_scope* to insert and/or look up symbols. Specifically, for our sample program, the *current_scope* can point either to the *global scope* or the *Alpha* scope.

We also need to update the semantic analyzer and set the *current_scope* to *None* in the constructor:

```
class SemanticAnalyzer(NodeVisitor):
    def __init__(self):
        self.current_scope = None
```

Clone the GitHub repository for the article (https://github.com/rspivak/lsbasi), run scope02.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope02.py) (it has all the changes we just discussed), inspect the output, and make sure you understand why every line is generated:

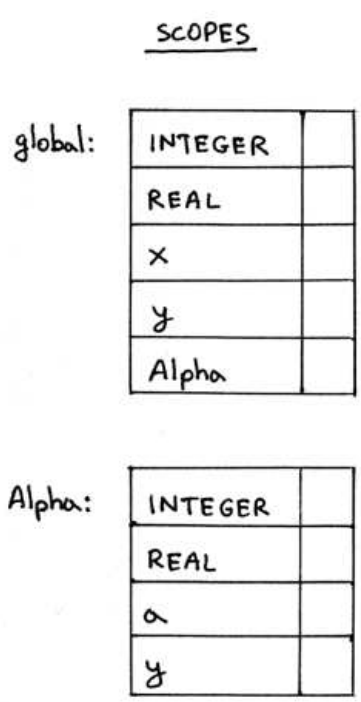
```
$ python scope02.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL
Insert: x
Lookup: REAL
Insert: y
Insert: Alpha
ENTER scope: Alpha
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: y


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : Alpha
Scope level    : 2
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: Alpha


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
  Alpha: <ProcedureSymbol(name=Alpha, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

Some things about the output above that I think are worth mentioning:

1. You can see that the two lines *Insert: INTEGER* and *Insert: REAL* are repeated twice in the output and the keys INTEGER and REAL are present in both scopes (scoped symbol tables): *global* and *Alpha*. The reason is that we create a separate scoped symbol table for every scope and the table initializes the built-in type symbols every time we create its instance. We'll change it later when we discuss nesting relationships and how they are expressed in code.

2. See how the line *Insert: Alpha* is printed before the line *ENTER scope: Alpha*. This is just a reminder that a name of a procedure is declared at a level that is one less than the level of the variables declared in the procedure itself.
3. You can see by inspecting the printed contents of the scoped symbol tables above what declarations they contain. See, for example, that *global scope* has the *Alpha* symbol in it.
4. From the contents of the *global scope* you can also see that the procedure symbol for the *Alpha* procedure also contains the procedure's formal parameters.

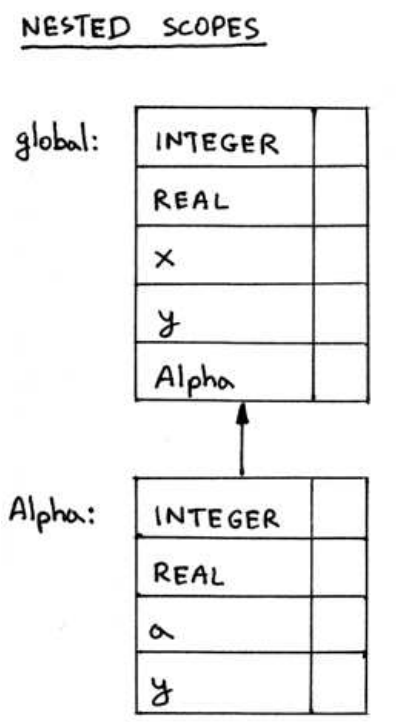
After we run the program, our scopes in memory would look something like this, just two separate scoped symbol tables:



Scope tree: Chaining scoped symbol tables

Okay, now every scope is represented by a separate scoped symbol table, but how do we represent the nesting relationship between the *global scope* and the scope *Alpha* as we showed in the nesting relationship diagram before? In other words, how do we express in code that the scope *Alpha* is nested within the *global scope*? The answer is chaining the tables together.

We'll chain the scoped symbol tables together by creating a link between them. In a way it'll be like a tree (we'll call it a *scope tree*), just an unusual one, because in this tree a child will be pointing to a parent, and not the other way around. Let's take a look the following *scope tree*:



In the *scope tree* above you can see that the scope *Alpha* is linked to the *global scope* by pointing to it. To put it differently, the scope *Alpha* is pointing to its *enclosing scope*, which is the *global scope*. It all means that the scope *Alpha* is nested within the *global scope*.

How do we implement scope chaining/linking? There are two steps:

1. We need to update the *ScopedSymbolTable* class and add a variable *enclosing_scope* that will hold a pointer to the scope's enclosing scope. This will be the link between scopes in the picture above.
2. We need to update the *visit_Program* and *visit_ProcedureDecl* methods to create an actual link to the scope's enclosing scope using the updated version of the *ScopedSymbolTable* class.

Let's start with updating the *ScopedSymbolTable* class and adding the *enclosing_scope* field. Let's also update the *__init__* and *__str__* methods. The *__init__* method will be modified to accept a new parameter, *enclosing_scope*, with the default value set to *None*. The *__str__* method will be updated to output the name of the enclosing scope. Here is the complete source code of the updated *ScopedSymbolTable* class:

```
class ScopedSymbolTable(object):
    def __init__(self, scope_name, scope_level, enclosing_scope=None):
        self._symbols = OrderedDict()
        self.scope_name = scope_name
        self.scope_level = scope_level
        self.enclosing_scope = enclosing_scope
        self._init_builtins()

    def _init_builtins(self):
        self.insert(BuiltinTypeSymbol('INTEGER'))
        self.insert(BuiltinTypeSymbol('REAL'))

    def __str__(self):
        h1 = 'SCOPE (SCOPED SYMBOL TABLE)'
        lines = ['\n', h1, '=' * len(h1)]
        for header_name, header_value in (
            ('Scope name', self.scope_name),
            ('Scope level', self.scope_level),
            ('Enclosing scope',
             self.enclosing_scope.scope_name if self.enclosing_scope else None
            )
        ):
            lines.append('%-15s: %s' % (header_name, header_value))
        h2 = 'Scope (Scoped symbol table) contents'
        lines.extend([h2, '-' * len(h2)])
        lines.extend(
            ('%7s: %r' % (key, value))
            for key, value in self._symbols.items()
        )
        lines.append('\n')
        s = '\n'.join(lines)
        return s

    __repr__ = __str__

    def insert(self, symbol):
        print('Insert: %s' % symbol.name)
        self._symbols[symbol.name] = symbol

    def lookup(self, name):
        print('Lookup: %s' % name)
        symbol = self._symbols.get(name)
        # 'symbol' is either an instance of the Symbol class or None
        return symbol
```

Now let's switch our attention to the *visit_Program* method:

```
def visit_Program(self, node):
    print('ENTER scope: global')
    global_scope = ScopedSymbolTable(
        scope_name='global',
        scope_level=1,
        enclosing_scope=self.current_scope, # None
    )
    self.current_scope = global_scope

    # visit subtree
    self.visit(node.block)

    print(global_scope)

    self.current_scope = self.current_scope.enclosing_scope
    print('LEAVE scope: global')
```

There are a couple of things here worth mentioning and repeating:

1. We explicitly pass the *self.current_scope* as the *enclosing_scope* argument when creating a scope
2. We assign the newly created global scope to the variable *self.current_scope*
3. We restore the variable *self.current_scope* to its previous value right before leaving the *Program* node. It’s important to restore the value of the *current_scope* after we’ve finished processing the node, otherwise the scope tree construction will be broken when we have more than two scopes in our program. We’ll see why shortly.

And, finally, let’s update the *visit_ProcedureDecl* method:

```
def visit_ProcedureDecl(self, node):
    proc_name = node.proc_name
    proc_symbol = ProcedureSymbol(proc_name)
    self.current_scope.insert(proc_symbol)

    print('ENTER scope: %s' %  proc_name)
    # Scope for parameters and local variables
    procedure_scope = ScopedSymbolTable(
        scope_name=proc_name,
        scope_level=self.current_scope.scope_level + 1,
        enclosing_scope=self.current_scope
    )
    self.current_scope = procedure_scope

    # Insert parameters into the procedure scope
    for param in node.params:
        param_type = self.current_scope.lookup(param.type_node.value)
        param_name = param.var_node.value
        var_symbol = VarSymbol(param_name, param_type)
        self.current_scope.insert(var_symbol)
        proc_symbol.params.append(var_symbol)

    self.visit(node.block_node)

    print(procedure_scope)

    self.current_scope = self.current_scope.enclosing_scope
    print('LEAVE scope: %s' %  proc_name)
```

Again, the main changes compared to the version in scope02.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope02.py) are:

1. We explicitly pass the *self.current_scope* as an *enclosing_scope* argument when creating a scope.
2. We no longer hard code the scope level of a procedure declaration because we can calculate the level automatically based on the scope level of the procedure's enclosing scope: it's the enclosing scope's level plus one.
3. We restore the value of the *self.current_scope* to its previous value (for our sample program the previous value would be the *global scope*) right before leaving the *ProcedureDecl* node.

Okay, let's see what the contents of the scoped symbol tables look like with the above changes. You can find all the changes in scope03a.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03a.py). Our sample program is:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
   begin

   end;

begin { Main }

end.  { Main }
```

Run scope03a.py on the command line and inspect the output:

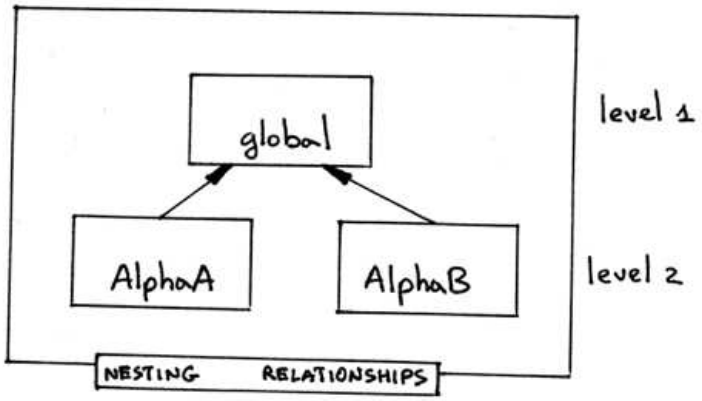
```
$ python scope03a.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL
Insert: x
Lookup: REAL
Insert: y
Insert: Alpha
ENTER scope: Alpha
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: y


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : Alpha
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: Alpha


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
  Alpha: <ProcedureSymbol(name=Alpha, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

You can see in the output above that the *global scope* doesn't have an enclosing scope and, the *Alpha*'s enclosing scope is the *global scope*, which is what we would expect, because the *Alpha* scope is nested within the *global scope*.

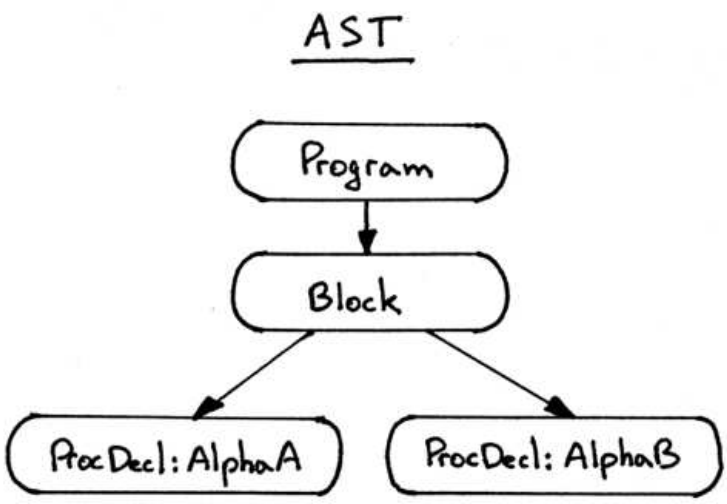
Now, as promised, let's consider why it is important to set and restore the value of the *self.current_scope* variable. Let's take a look at the following program, where we have two procedure declarations in the *global scope*:

```
program Main;
   var x, y : real;

   procedure AlphaA(a : integer);
      var y : integer;
   begin { AlphaA }

   end;  { AlphaA }

   procedure AlphaB(a : integer);
      var b : integer;
   begin { AlphaB }

   end;  { AlphaB }

begin { Main }

end.  { Main }
```

The nesting relationship diagram for the sample program looks like this:



An AST for the program (I left only the nodes that are relevant to this example) is something like this:



If we don’t restore the current scope when we leave the *Program* and *ProcedureDecl* nodes what is going to happen? Let’s see.

The way our semantic analyzer walks the tree is depth first, left-to-right, so it will traverse the *ProcedureDecl* node for *AlphaA* first and then it will visit the *ProcedureDecl* node for *AlphaB*. The problem here is that if we don’t restore the *self.current_scope* before leaving *AlphaA* the *self.current_scope* will be left pointing to *AlphaA* instead of the *global scope* and, as a result, the semantic analyzer will create the scope *AlphaB* at level 3, as if it was nested within the scope *AlphaA*, which is, of course, incorrect.

To see the broken behavior when the current scope is not being restored before leaving *Program* and/or *ProcedureDecl* nodes, download and run the scope03b.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03b.py) on the command line:

```
$ python scope03b.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL
Insert: x
Lookup: REAL
Insert: y
Insert: AlphaA
ENTER scope: AlphaA
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: y


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : AlphaA
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: AlphaA
Insert: AlphaB
ENTER scope: AlphaB
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: b


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : AlphaB
Scope level    : 3
Enclosing scope: AlphaA
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      b: <VarSymbol(name='b', type='INTEGER')>


LEAVE scope: AlphaB
```

```
SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
 AlphaA: <ProcedureSymbol(name=AlphaA, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

As you can see, scope tree construction in our semantic analyzer is completely broken in the presence of more than two scopes:

1. Instead of two scope levels as shown in the nesting relationships diagram, we have three levels
2. The *global scope* contents doesn't have *AlphaB* in it, only *AlphaA*.

To construct a scope tree correctly, we need to follow a really simple procedure:

1. When we ENTER a *Program* or *ProcedureDecl* node, we create a new scope and assign it to the *self.current_scope*.
2. When we are about to LEAVE the *Program* or *ProcedureDecl* node, we restore the value of the *self.current_scope*.

You can think of the *self.current_scope* as a stack pointer and a *scope tree* as a collection of stacks:

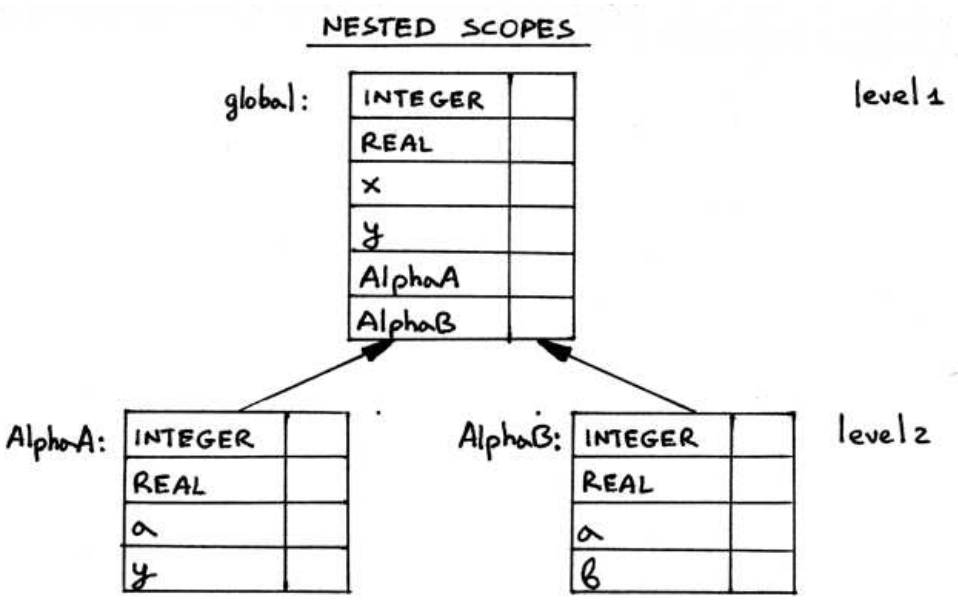
1. When you visit a *Program* or *ProcedureDecl* node, you push a new scope on the stack and adjust the stack pointer *self.current_scope* to point to the top of stack, which is now the most recently pushed scope.
2. When you are about to leave the node, you pop the scope off the stack and you also adjust the stack pointer to point to the previous scope on the stack, which is now the new top of stack.

To see the correct behavior in the presence of multiple scopes, download and run scope03c.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03c.py) on the command line. Study the output. Make sure you understand what is going on:

```
$ python scope03c.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL
Insert: x
Lookup: REAL
Insert: y
Insert: AlphaA
ENTER scope: AlphaA
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: y


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : AlphaA
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: AlphaA
Insert: AlphaB
ENTER scope: AlphaB
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: b


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : AlphaB
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      b: <VarSymbol(name='b', type='INTEGER')>


LEAVE scope: AlphaB
```

```
SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
 AlphaA: <ProcedureSymbol(name=AlphaA, parameters=[<VarSymbol(name='a', type='INTEGER')>])>
 AlphaB: <ProcedureSymbol(name=AlphaB, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

This is how our scoped symbol tables look like after we've run scope03c.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03c.py) and correctly constructed the *scope tree*:



Again, as I've mentioned above, you can think of the scope tree above as a collection of scope stacks.

Now let's continue and talk about how *name resolution* works when we have nested scopes.

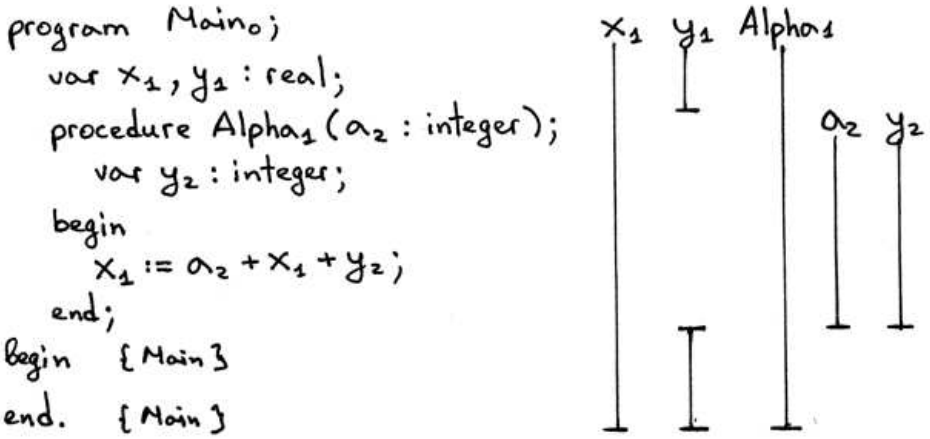
Nested scopes and name resolution

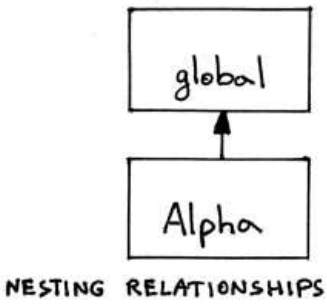
Our focus before was on variable and procedure declarations. Let's add variable references to the mix.

Here is a sample program with some variable references in it:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
   begin
      x := a + x + y;
   end;

begin { Main }

end.  { Main }
```

Or visually with some additional information:



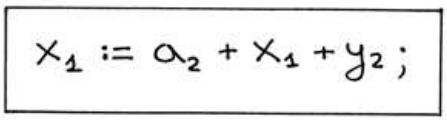


NESTING RELATIONSHIPS

Scope level	Scope name	Names declared in each scope
1	global	x, y, Alpha, INTEGER, REAL
2	Alpha	a, y

SCOPE INFORMATION TABLE

Let's turn our attention to the assignment statement **x := a + x + y;** Here it is with subscripts:

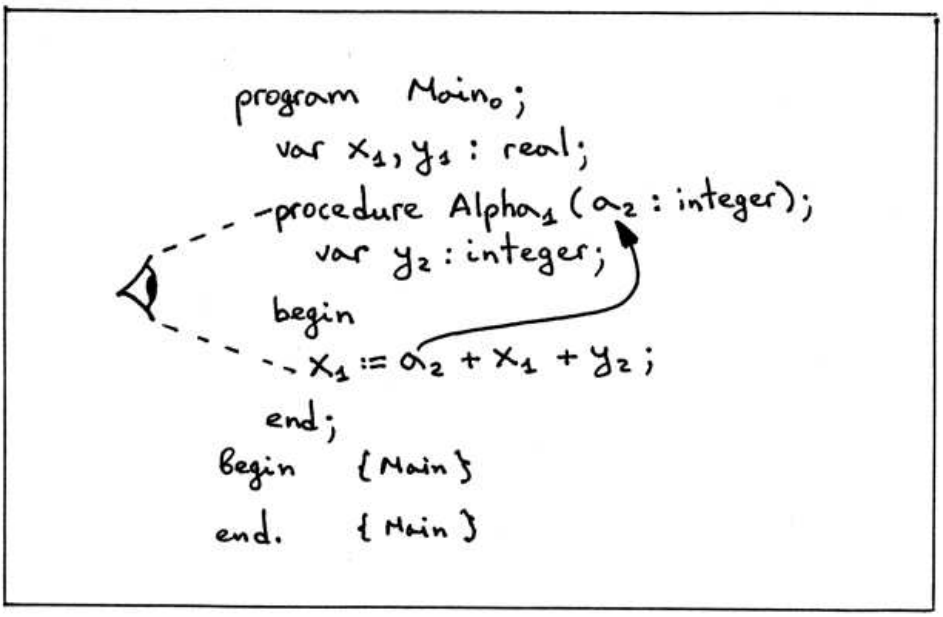


We see that **x** resolves to a declaration at level 1, **a** resolves to a declaration at level 2 and **y** also resolves to a declaration at level 2. How does that resolution work? Let's see how.

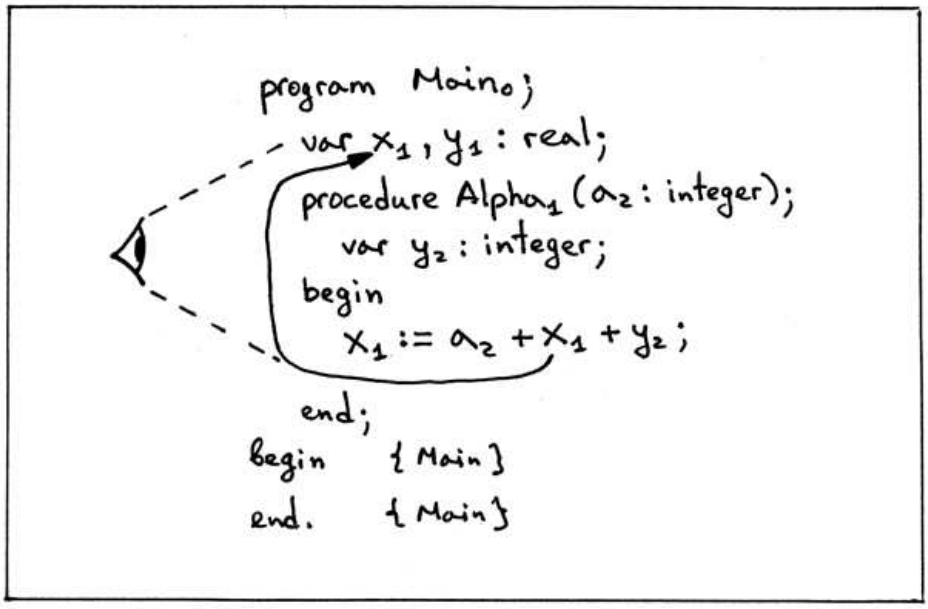
Lexically (statically) scoped languages like Pascal follow ***the most closely nested scope*** rule when it comes to name resolution. It means that, in every scope, a name refers to its lexically closest declaration. For our assignment statement, let's go over every variable reference and see how the rule works in practice:

1. Because our semantic analyzer visits the right-hand side of the assignment first, we'll start with the variable reference **a** from the arithmetic expression **a + x + y**. We begin our search for **a**'s declaration in the lexically closest scope, which is the *Alpha* scope. The *Alpha* scope contains variable declarations in the *Alpha* procedure including the procedure's formal parameters. We

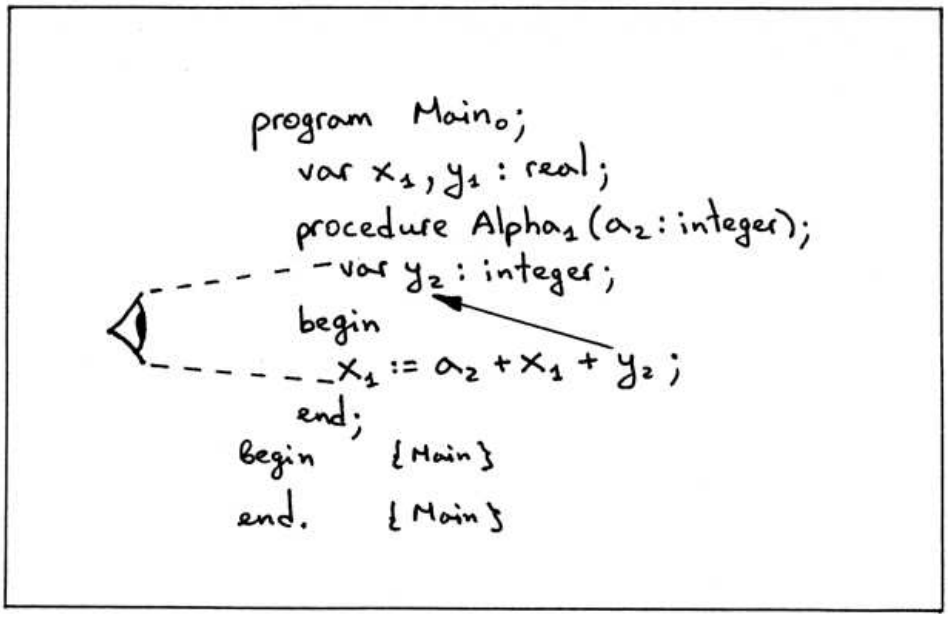
find the declaration of **a** in the *Alpha* scope: it's the formal parameter **a** of the *Alpha* procedure - a variable symbol that has type **integer**. We usually do the search by scanning the source code with our eyes when resolving names (remember, **a2** is not the name of a variable, 2 is the subscript here, the variable name is **a**):



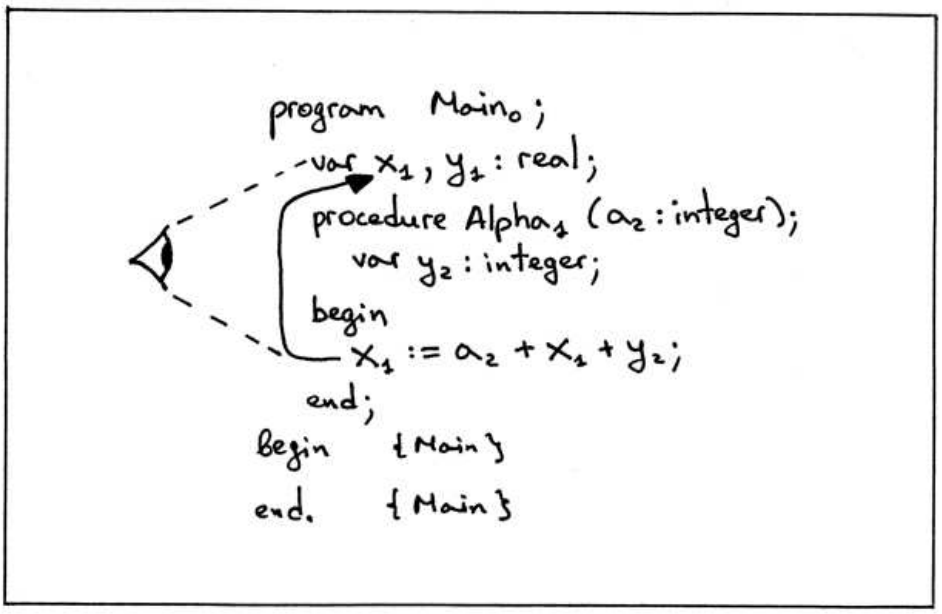
2. Now onto the variable reference **x** from the arithmetic expression **a + x + y**. Again, first we search for the declaration of **x** in the lexically closest scope. The lexically closest scope is the *Alpha* scope at level 2. The scope contains declarations in the *Alpha* procedure including the procedure's formal parameters. We don't find **x** at this scope level (in the *Alpha* scope), so we go up the chain to the *global scope* and continue our search there. Our search succeeds because the *global scope* has a variable symbol with the name **x** in it:



3. Now, let's look at the variable reference **y** from the arithmetic expression **a + x + y**. We find its declaration in the lexically closest scope, which is the *Alpha* scope. In the *Alpha* scope the variable **y** has type **integer** (if there weren't a declaration for **y** in the *Alpha* scope we would scan the text and find **y** in the outer/global scope and it would have **real** type in that case):



4. And, finally, the variable **x** from the left hand side of the assignment statement **x := a + x + y;** It resolves to the same declaration as the variable reference **x** in the arithmetic expression on the right-hand side:



How do we implement that behavior of looking in the current scope, and then looking in the enclosing scope, and so on until we either find the symbol we're looking for or we've reached the top of the scope tree and there are no more scopes left? We simply need to extend the *lookup* method in the *ScopedSymbolTable* class to continue its search up the chain in the scope tree:

```
def lookup(self, name):
    print('Lookup: %s. (Scope name: %s)' % (name, self.scope_name))
    # 'symbol' is either an instance of the Symbol class or None
    symbol = self._symbols.get(name)

    if symbol is not None:
        return symbol

    # recursively go up the chain and lookup the name
    if self.enclosing_scope is not None:
        return self.enclosing_scope.lookup(name)
```

The way the updated *lookup* method works:

1. Search for a symbol by name in the current scope. If the symbol is found, then return it.
2. If the symbol is not found, recursively traverse the tree and search for the symbol in the scopes up the chain. You don’t have to do the lookup recursively, you can rewrite it into an iterative form; the important part is to follow the link from a nested scope to its enclosing scope and search for the symbol there and up the tree until either the symbol is found or there are no more scopes left because you’ve reached the top of the scope tree.
3. The *lookup* method also prints the scope name, in parenthesis, where the lookup happens to make it clearer that lookup goes up the chain to search for a symbol, if it can’t find it in the current scope.

Let’s see what our semantic analyzer outputs for our sample program now that we’ve modified the way the *lookup* searches the scope tree for a symbol. Download scope04a.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope04a.py) and run it on the command line:

```
$ python scope04a.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL. (Scope name: global)
Insert: x
Lookup: REAL. (Scope name: global)
Insert: y
Insert: Alpha
ENTER scope: Alpha
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: a
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: y
Lookup: a. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)
Lookup: y. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : Alpha
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
-----------------------------------
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: Alpha


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
  Alpha: <ProcedureSymbol(name=Alpha, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

Inspect the output above and pay attention to the *ENTER* and *Lookup* messages. A couple of things worth mentioning here:

1. Notice how the semantic analyzer looks up the *INTEGER* built-in type symbol before inserting the variable symbol **a**. It searches *INTEGER* first in the current scope, *Alpha*, doesn't find it, then goes up the tree all the way to the *global scope*, and finds the symbol there:

```
ENTER scope: Alpha
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: a
```

2. Notice also how the analyzer resolves variable references from the assignment statement **x := a + x + y**:

```
Lookup: a. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)
Lookup: y. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)
```

   The analyzer starts its search in the current scope and then goes up the tree all the way to the *global scope*.

Let's also see what happens when a Pascal program has a variable reference that doesn't resolve to a variable declaration as in the sample program below:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
   begin
      x := b + x + y; { ERROR here! }
   end;

begin { Main }

end.  { Main }
```

Download scope04b.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope04b.py) and run it on the command line:

```
$ python scope04b.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL. (Scope name: global)
Insert: x
Lookup: REAL. (Scope name: global)
Insert: y
Insert: Alpha
ENTER scope: Alpha
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: a
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: y
Lookup: b. (Scope name: Alpha)
Lookup: b. (Scope name: global)
Error: Symbol(identifier) not found 'b'
```

As you can see, the analyzer tried to resolve the variable reference **b** and searched for it in the *Alpha* scope first, then the *global scope*, and, not being able to find a symbol with the name **b**, it threw the semantic error.

Okay great, now we know how to write a semantic analyzer that can analyze a program for semantic errors when the program has nested scopes.

Source-to-source compiler

Now, onto something completely different. Let's write a *source-to-source compiler*! Why would we do it? Aren't we talking about interpreters and nested scopes? Yes, we are, but let me explain why I think it might be a good idea to learn how to write a source-to-source compiler right now.

First, let's talk about definitions. What is *a source-to-source compiler*? For the purpose of this article, let's define a ***source-to-source compiler*** as a compiler that translates a program in some source language into a program in the same (or almost the same) source language.

So, if you write a translator that takes as an input a Pascal program and outputs a Pascal program, possibly modified, or enhanced, the translator in this case is called a *source-to-source compiler*.

A good example of a source-to-source compiler for us to study would be a compiler that takes a Pascal program as an input and outputs a Pascal-like program where every name is subscripted with a corresponding scope level, and, in addition to that, every variable reference also has a type indicator. So we want a source-to-source compiler that would take the following Pascal program:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
   begin
      x := a + x + y;
   end;

begin { Main }

end.  { Main }
```

and turn it into the following Pascal-like program:

```
program Main0;
   var x1 : REAL;
   var y1 : REAL;
   procedure Alpha1(a2 : INTEGER);
      var y2 : INTEGER;

   begin
      <x1:REAL> := <a2:INTEGER> + <x1:REAL> + <y2:INTEGER>;
   end; {END OF Alpha}

begin

end. {END OF Main}
```

Here is the list of modifications our source-to-source compiler should make to an input Pascal program:

1. Every declaration should be printed on a separate line, so if we have multiple declarations in the input Pascal program, the compiled output should have each declaration on a separate line. We can see in the text above, for example, how the line *var x, y : real;* gets converted into multiple lines.
2. Every name should get subscripted with a number corresponding to the scope level of the respective declaration.
3. Every variable reference, in addition to being subscripted, should also be printed in the following form: *<var_name_with_subscript:type>*
4. The compiler should also add a comment at the end of every block in the form *{END OF … }*, where the ellipses will get substituted either with a program name or procedure name. That will help us identify the textual boundaries of procedures faster.

As you can see from the generated output above, this source-to-source compiler could be a useful tool for understanding how name resolution works, especially when a program has nested scopes, because the output generated by the compiler would allow us to quickly see to what declaration and in what scope a certain variable reference resolves to. This is good help when learning about symbols, nested scopes, and name resolution.

How can we implement a source-to-source compiler like that? We have actually covered all the necessary parts to do it. All we need to do now is extend our semantic analyzer a bit to generate the enhanced output. You can see the full source code of the compiler here

(https://github.com/rspivak/lsbasi/blob/master/part14/src2srccompiler.py). It is basically a semantic analyzer on drugs, modified to generate and return strings for certain AST nodes.

Download src2srccompiler.py (https://github.com/rspivak/lsbasi/blob/master/part14/src2srccompiler.py), study it, and experiment with it by passing it different Pascal programs as an input.

For the following program, for example:

```
program Main;
   var x, y : real;
   var z : integer;

   procedure AlphaA(a : integer);
      var y : integer;
   begin { AlphaA }
      x := a + x + y;
   end;  { AlphaA }

   procedure AlphaB(a : integer);
      var b : integer;
   begin { AlphaB }
   end;  { AlphaB }

begin { Main }
end.  { Main }
```

The compiler generates the following output:

```
$ python src2srccompiler.py nestedscopes03.pas
program Main0;
   var x1 : REAL;
   var y1 : REAL;
   var z1 : INTEGER;
   procedure AlphaA1(a2 : INTEGER);
      var y2 : INTEGER;

   begin
      <x1:REAL> := <a2:INTEGER> + <x1:REAL> + <y2:INTEGER>;
   end; {END OF AlphaA}
   procedure AlphaB1(a2 : INTEGER);
      var b2 : INTEGER;

   begin

   end; {END OF AlphaB}

begin

end. {END OF Main}
```

Cool beans and congratulations, now you know how to write a basic source-to-source compiler!

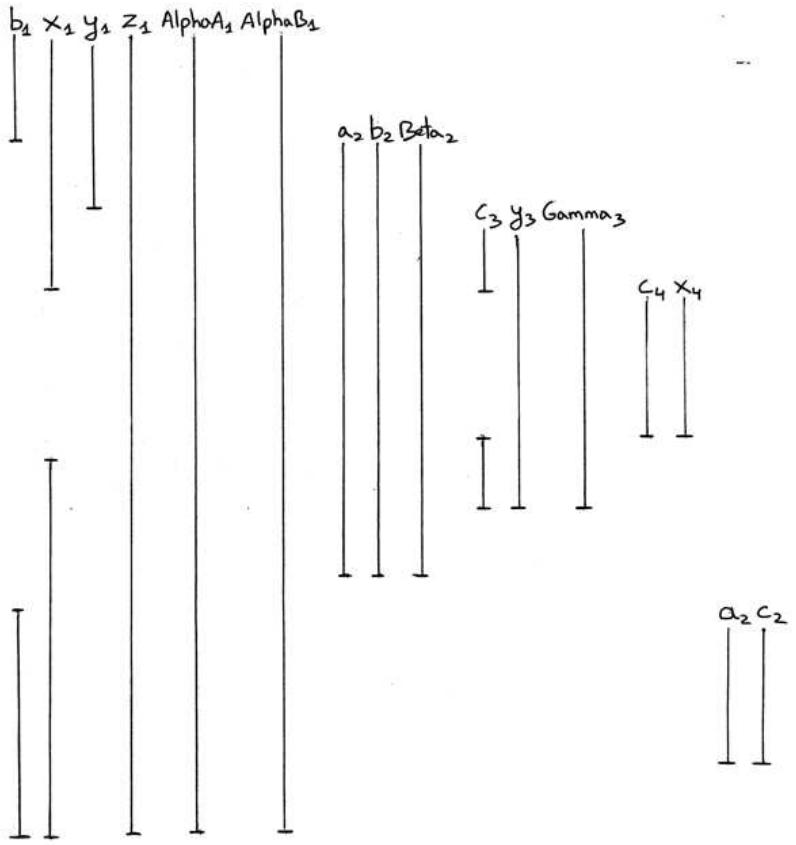
Use it to further your understanding of nested scopes, name resolution, and what you can do when you have an AST and some extra information about the program in the form of symbol tables.

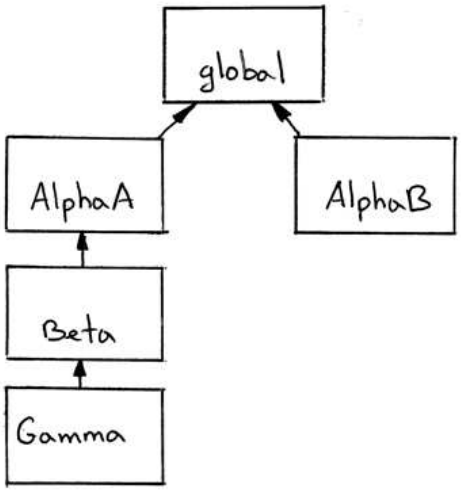
Now that we have a useful tool to subscript our programs for us, let's take a look at a bigger example of nested scopes that you can find in nestedscopes04.pas (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes04.pas):

```
program Main;
   var b, x, y : real;
   var z : integer;

   procedure AlphaA(a : integer);
      var b : integer;

      procedure Beta(c : integer);
         var y : integer;

         procedure Gamma(c : integer);
            var x : integer;
         begin { Gamma }
            x := a + b + c + x + y + z;
         end;  { Gamma }

      begin { Beta }

      end;  { Beta }

   begin { AlphaA }

   end;  { AlphaA }

   procedure AlphaB(a : integer);
      var c : real;
   begin { AlphaB }
      c := a + b;
   end;  { AlphaB }

begin { Main }
end.  { Main }
```

Below you can see the declarations' scopes, nesting relationships diagram, and scope information table:

```
program Main0;
  var b1, x1, y1 : real;
  var z1 : integer;
  procedure AlphaA1 (a2 : integer);
    var b2 : integer;
    procedure Beta2 (c3 : integer);
      var y3 : integer;
      procedure Gamma3 (c4 : integer);
        var x4 : integer;
      begin
        x4 := a2 + b2 + c4 + x4 + y3 + z1;
      end; { Gamma }
    begin
    end; { Beta }
  begin
  end; { AlphaA }
  procedure AlphaB1 (a2 : integer);
    var c2 : real;
  begin
    c2 := a2 + b1;
  end; { AlphaB }
begin
end. { Main }
```





NESTING RELATIONSHIPS

Scope level	Scope name	Names declared in each scope
1	global	b, x, y, z, AlphaA, AlphaB, INTEGER, REAL
2	AlphaA	a, b, Beta
2	AlphaB	a, c
3	Beta	c, y, Gamma
4	Gamma	c, x

SCOPE INFORMATION TABLE

Let's run our source-to-source compiler and inspect the output. The subscripts should match the ones in the scope information table in the picture above:

```
$ python src2srccompiler.py nestedscopes04.pas
program Main0;
   var b1 : REAL;
   var x1 : REAL;
   var y1 : REAL;
   var z1 : INTEGER;
   procedure AlphaA1(a2 : INTEGER);
      var b2 : INTEGER;
      procedure Beta2(c3 : INTEGER);
         var y3 : INTEGER;
         procedure Gamma3(c4 : INTEGER);
            var x4 : INTEGER;

         begin
            <x4:INTEGER> := <a2:INTEGER> + <b2:INTEGER> + <c4:INTEGER> + <x4:INTEGER> + <y3:INTE
         end; {END OF Gamma}

      begin

      end; {END OF Beta}

   begin

   end; {END OF AlphaA}
   procedure AlphaB1(a2 : INTEGER);
      var c2 : REAL;

   begin
      <c2:REAL> := <a2:INTEGER> + <b1:REAL>;
   end; {END OF AlphaB}

begin

end. {END OF Main}
```

Spend some time studying both the pictures and the output of the source-to-source compiler. Make sure you understand the following main points:

- The way the vertical lines are drawn to show the scope of the declarations.
- That a hole in a scope indicates that a variable is re-declared in a nested scope.
- That *AlphaA* and *AlphaB* are declared in the global scope.
- That *AlphaA* and *AlphaB* declarations introduce new scopes.
- How scopes are nested within each other, and their nesting relationships.
- Why different names, including variable references in assignment statements, are subscripted the way they are. In other words, how name resolution and specifically the *lookup* method of chained scoped symbol tables works.

Also run the following program (https://github.com/rspivak/lsbasi/blob/master/part14/scope05.py):

```
$ python scope05.py nestedscopes04.pas
```

and inspect the contents of the chained scoped symbol tables and compare it with what you see in the scope information table in the picture above. And don't forget about the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part14/genastdot.py), which you can use to generate a

visual diagram of an AST to see how procedures are nested within each other in the tree.

Before we wrap up our discussion of nested scopes for today, recall that earlier we removed the semantic check that was checking source programs for duplicate identifiers. Let's put it back. For the check to work in the presence of nested scopes and the new behavior of the *lookup* method, though, we need to make some changes. First, we need to update the *lookup* method and add an extra parameter that will allow us to limit our search to the current scope only:

```
def lookup(self, name, current_scope_only=False):
    print('Lookup: %s. (Scope name: %s)' % (name, self.scope_name))
    # 'symbol' is either an instance of the Symbol class or None
    symbol = self._symbols.get(name)

    if symbol is not None:
        return symbol

    if current_scope_only:
        return None

    # recursively go up the chain and lookup the name
    if self.enclosing_scope is not None:
        return self.enclosing_scope.lookup(name)
```

And second, we need to modify the *visit_VarDecl* method and add the check using our new *current_scope_only* parameter in the *lookup* method:

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.current_scope.lookup(type_name)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)

    # Signal an error if the table alrady has a symbol
    # with the same name
    if self.current_scope.lookup(var_name, current_scope_only=True):
        raise Exception(
            "Error: Duplicate identifier '%s' found" % var_name
        )

    self.current_scope.insert(var_symbol)
```

If we don't limit the search for a duplicate identifier to the current scope, the lookup might find a variable symbol with the same name in an outer scope and, as a result, would throw an error, while in reality there was no semantic error to begin with.

Here is the output from running scope05.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope05.py) with a program that doesn't have duplicate identifier errors. You can notice below that the output has more lines in it, due to our duplicate identifier check that looks up for a duplicate name before inserting a new symbol:

```
$ python scope05.py nestedscopes02.pas
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL. (Scope name: global)
Lookup: x. (Scope name: global)
Insert: x
Lookup: REAL. (Scope name: global)
Lookup: y. (Scope name: global)
Insert: y
Insert: Alpha
ENTER scope: Alpha
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: a
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Lookup: y. (Scope name: Alpha)
Insert: y
Lookup: a. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)
Lookup: y. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : Alpha
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
------------------------------------
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: Alpha


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
  Alpha: <ProcedureSymbol(name=Alpha, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

Now, let's take scope05.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope05.py) for another test drive and see how it catches a duplicate identifier semantic error.

For example, for the following erroneous program (https://github.com/rspivak/lsbasi/blob/master/part14/dupiderror.pas) with a duplicate declaration of ***a*** in the *Alpha* scope:

```
program Main;
   var x, y: real;

   procedure Alpha(a : integer);
      var y : integer;
      var a : real;  { ERROR here! }
   begin
      x := a + x + y;
   end;

begin { Main }

end.  { Main }
```

the program generates the following output:

```
$ python scope05.py dupiderror.pas
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL. (Scope name: global)
Lookup: x. (Scope name: global)
Insert: x
Lookup: REAL. (Scope name: global)
Lookup: y. (Scope name: global)
Insert: y
Insert: Alpha
ENTER scope: Alpha
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: a
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Lookup: y. (Scope name: Alpha)
Insert: y
Lookup: REAL. (Scope name: Alpha)
Lookup: REAL. (Scope name: global)
Lookup: a. (Scope name: Alpha)
Error: Duplicate identifier 'a' found
```

It caught the error as expected.

On this positive note, let's wrap up our discussion of scopes, scoped symbol tables, and nested scopes for today.

Summary

We've covered a lot of ground. Let's quickly recap what we learned in this article:

- We learned about *scopes*, why they are useful, and how to implement them in code.
- We learned about *nested scopes* and how *chained scoped symbol tables* are used to implement nested scopes.
- We learned how to code a semantic analyzer that walks an AST, builds *scoped symbols tables*, chains them together, and does various semantic checks.
- We learned about *name resolution* and how the semantic analyzer resolves names to their declarations using *chained scoped symbol tables (scopes)* and how the *lookup* method recursively goes up the chain in a *scope tree* to find a declaration corresponding to a certain name.
- We learned that building a *scope tree* in the semantic analyzer involves walking an AST, "pushing" a new scope on top of a scoped symbol table stack when ENTERing a certain AST node and "popping" the scope off the stack when LEAVing the node, making a *scope tree* look like a collection of scoped symbol table stacks.
- We learned how to write a *source-to-source compiler*, which can be a useful tool when learning about nested scopes, scope levels, and name resolution.

Exercises

Time for exercises, oh yeah!

1. You've seen in the pictures throughout the article that the *Main* name in a program statement had subscript zero. I also mentioned that the program's name is not in the *global scope* and it's in some other outer scope that has level zero. Extend spi.py (https://github.com/rspivak/lsbasi/blob/master/part14/spi.py) and create a *builtins* scope, a new scope at level 0, and move the built-in types INTEGER and REAL into that scope. For fun and practice, you can also update the code to put the program name into that scope as well.
2. For the source program in nestedscopes04.pas (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes04.pas) do the following:
    1. Write down the source Pascal program on a piece of paper
    2. Subscript every name in the program indicating the scope level of the declaration the name resolves to.
    3. Draw vertical lines for every name declaration (variable and procedure) to visually show its scope. Don't forget about scope holes and their meaning when drawing.

    4. Write a source-to-source compiler for the program without looking at the example source-to-source compiler in this article.
    5. Use the original src2srccompiler.py (https://github.com/rspivak/lsbasi/blob/master/part14/src2srccompiler.py) program to verify the output from your compiler and whether you subscripted the names correctly in the exercise (2.2).
3. Modify the source-to-source compiler to add subscripts to the built-in types INTEGER and REAL
4. Uncomment the following block in the spi.py (https://github.com/rspivak/lsbasi/blob/master/part14/spi.py)

    ```
    # interpreter = Interpreter(tree)
    # result = interpreter.interpret()
    # print('')
    # print('Run-time GLOBAL_MEMORY contents:')
    # for k, v in sorted(interpreter.GLOBAL_MEMORY.items()):
    #     print('%s = %s' % (k, v))
    ```

    Run the interpreter with the part10.pas (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas) file as an input:

    ```
    $ python spi.py part10.pas
    ```

    Spot the problems and add the missing methods to the semantic analyzer.

That's it for today. In the next article we'll learn about runtime, call stack, implement procedure calls, and write our first version of a recursive factorial function. Stay tuned and see you soon!

If you're interested, here is a list of books (affiliate links) I referred to most when preparing the article:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (https://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=5d5ca8c07bff5452ea443d8319e7703d)
2. Engineering a Compiler, Second Edition (https://www.amazon.com/gp/product/012088478X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=012088478X&linkCode=as2&tag=russblo0b-20&linkId=74578959d7d04bee4050c7bff1b7d02e)
3. Programming Language Pragmatics, Fourth Edition (https://www.amazon.com/gp/product/0124104096/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0124104096&linkCode=as2&tag=russblo0b-20&linkId=8db1da254b12fe6da1379957dda717fc)
4. Compilers: Principles, Techniques, and Tools (2nd Edition) (https://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=31743d76157ef1377153dba78c54e177)